第一章 绪 论

本章学习重点

本章介绍C语言的发展历史以及C语言的特点，然后通过简单例子展示C语言的风貌，最后告诉你如何在计算机上运行C程序。

第一节 C语言的产生及特点

C语言是一种得到广泛重视并普遍应用的计算机程序设计语言，也是国际上公认的最重要的几种通用程序设计语言之一。它适合作系统描述语言，既可用来写系统软件，也可用来写应用软件。

C语言是美国贝尔实验室的Dennis. M. Ritchie于1972年设计实现的。C语言直接来源于B语言，但它的根源可以追溯到ALGOL60。ALGOL60结构严谨，其设计者非常注重语法、分程序结构，因此对于后来许多重要的程序设计语言，如PASCAL，PL/I，SIMULA67产生过重要的影响。但它是面向过程的语言，与计算机硬件相距甚远，不适合编写系统软件。1963年英国剑桥大学在ALGOL60的基础上推出更接近硬件的CPL语言，但CPL太复杂，难于实现。1967年，剑桥大学的Matin Rinchards对CPL语言作了简化，推出了BCPL语言。1970年贝尔实验室的Ken Thompson以BCPL为基础，设计了更简单也更接近硬件的B语言(取BCPL的第一个字母)。B语言是一种解释性语言，功能上也不够强，为了很好地适应系统程序设计的要求，Ritchie把B发展成称之为C的语言(取BCPL的第二个字母)。C语言既保持了BCPL和B的优点(如精练，接近硬件)，又克服了它们的缺点(如过于简单，数据无类型等)。1973年K. Thompson和D. M. Ritchie用C改写了UNIX代码，并在PDP－11计算机上加以实现，即UNIX版本5，这一版本奠定了UNIX系统的基础，使UNIX逐渐成为最重要的操作系统之一。

C语言是D. M. Ritchie1972年设计实现的

图1-1展示了C语言的由来。

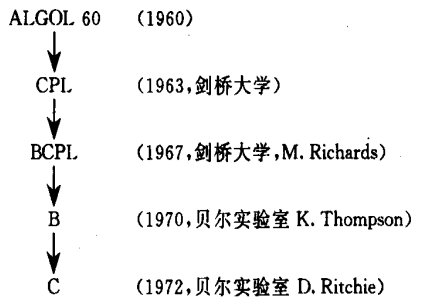


图1-1 C语言的由来

C 设计语言的目的是为描述和实现 UNIX 操作系统提供一种工具语言。但 C 并没有被缚束在任何特定的硬件或操作系统上，它具有良好的可移植性。1977 年出现了不依赖于具体机器的 C 语言编译文本，使向各种机器移植 C 变得更加简单，这也推动了 UNIX 系统的广泛实现。随着 UNIX 的日益普及，又反过头来带动了 C 语言的迅速推广，使它先后被移植到各种大、中、小、微型计算机上。

1978 年，贝尔实验室的 Brian. W. Kernighan 和 Dennis. M. Ritchie（合称 K&R）合著了《The C Programming Language》一书，并在附录中提供了 C 语言参考手册，这本书成为以后广泛使用的 C 语言的基础，被人们称作非官方的 C 语言标准。1983 年美国国家标准化协会（ANSI）开始制定新的标准，这就是 ANSI C 标准。1990 年，C 语言成为国际标准化组织（ISO）通过的标准语言。

C 语言是作为描述系统的语言而设计，但随着其日益广泛的应用，特别是 80 年代以后各种微机 C 语言的普及，它已经成为众多程序员最喜爱的语言，它的使用覆盖了几乎计算机的所有领域，包括操作系统、编译程序、数据库管理程序、CAD、过程控制、图形图象处理等等。

C 语言如此成功是有其自身特点的：

1. C 语言是比较“低级的”语言。

有人把 C 语言称为“高级语言中的低级语言”，也有人称它是“中级语言”。这样说不是说它功能差或难于使用，而是指它具有许多通常只有象汇编语言才具备的功能，如位操作、直接访问物理地址等等，这使 C 语言在进行系统程序设计时显得非常有效，而过去系统软件通常只能用汇编语言编写。事实上，C 语言的许多应用场合是汇编语言的传统领地，现在用 C 来代替汇编，使程序员得以减轻负担、提高效率，而写出的程序具有更好的可移植性。

C 语言具有更多的接近硬件操作的功能，而不提供直接处理复合对象，如作为整体看待的字符串、数组等的操作，这些较高级的功能必须通过显式调用函数来完成。这看起来是个缺陷，但这使语言的规模较小，更容易说明，学习起来也快。比如说，C 语言只有 32 个关键字，而一些微机上的 BASIC，关键字多达 100 个以上。

2. C 语言是结构化的语言。

C 语言在结构上类似于 ALGOL60、PASCAL 等结构化语言。这大概与它源于 ALGOL60 以及 60 年代末 70 年代初结构化程序设计方法的兴起有关。C 语言的主要结构成分是函数——C 的子程序。函数允许一个程序中的各任务分别定义和编码，使程序模块化，在函数的外部只需了解函数的功能，而将实现的细节隐藏起来，设计得好的函数能够正确地工作而对程序的其他部分不产生副作用。C 语言还提供了多种结构化的控制语句，如用于循环的 for、while、do-while 语句，用于判定的 if-else、switch 语句等，以满足结构化程序设计的要求。

3. C 语言是程序员的语言。

看到这个提法，你可能会奇怪：难道还有不是程序员的程序设计语言吗？确实有许多程序设计语言不是为专业程序员设计的，比如说：FORTRAN 是为工程师，COBOL 是为商业人员，PASCAL 是为学生，而 BASIC 是为非程序员设计的。C 是为专业程序员设计的语言。Ritchie 是专业程序员，而 C 最初是为了他自己写 UNIX 操作系统而设计的。C 语言实现了程序员的期望：很少限制，很少强求，程序设计自由度大，方便的控制结构，独立的函数，紧凑的关键字集合和较高的执行效率。用 C 编写程序可获得高效的机器代码，其效率通常只比

汇编语言生成的机器代码低10～20%，而同时C又具有PASCAL那样的结构，这就难怪有大量的程序员喜欢它。

C语言的语法限制不太严，例如，对数组下标越界不做检查，整型、字符型数据可以通用，不专设逻辑型数据而以整型来代替等。较少的限制给程序员带来较大自由，这就要求程序员在编程时应确实明白自己在做什么，而不要把检查错误的工作仅寄托于编译程序。当然这会带来一些麻烦，但作为程序员应该考虑好再开始编码。其实，有时候让计算机惩罚一下粗枝大叶的程序员也不一定是什么坏事。

C语言得到广泛使用的主要原因可能就是因为程序员喜欢它。但C也不是专业人员的专利，非专业人员经过实践也会熟练地掌握它。现在有许多不同专业的非计算机专业人员正在使用或正在转向C语言。C语言是我们大家共同的财富。

第二节　用C语言编写程序

学习一种程序设计语言唯一的有效途径就是用它编写程序。下面几个简单的程序将带你进入C的世界，开始我们的学习。

第一个程序总是输出一个字符串，这里，选择著名的"hello, world"程序，以表示对K&R的敬意。

例1.1

```
main()
{
    printf("hello,world\n");
}
```

这个程序的运行结果是输出字符串

```
hello,world
```

上面的程序称C语言源程序，简称C程序。为了让它运行，必须先找一个编辑程序输入它，然后编译，装入，最后执行。

现在让我们来看看程序本身。main是一个函数名，表示"主函数"。一个C程序，不管简单或复杂，总是由一个或多个函数组成，由这些函数实现要做的操作。函数名可以按照你的喜好去取，但main是一个特殊的名字。C程序总是从main函数开始执行，这意味着，一个程序必须在某个地方有一个main函数。

花括号{}括起构造函数的语句，称函数体。函数体中只有一条函数调用语句

```
printf("hello,world\n");
```

其功能是将双引号""中的内容输出。printf是C语言中的库函数，它来自计算机厂家提供的输入输出库，你可以直接去调用它。双引号中的内容是调用printf时提供的参数，表示为一个字符串。字符串中的\n是C语言中的换行字符，在输出时，它将终端的光标移动到下一行的开始。如果不使用\n，输出时就会发现，光标停在输出字符串的后面。printf不提供自动换行的功能，每当在程序中需要执行换行操作时，都必须写出\n。例如，我们把例1.1改写一下，

```
main()
{
    printf("hello,") ;
    printf("world") ;
    printf("\n") ;
}
```

执行结果也是一样的。如果要将程序改成

```
main()
{
    printf("hello,\n") ;
    printf("world\n") ;
}
```

你会发现输出变为：

```
hello,
world
```

在这里我们可以看到，C 语言倾向于把怎样处理的权利交给程序员。

printf 语句的最后跟着一个分号“;”，它表明一个语句的结束。

例 1.2

```
/*求两个整数之和*/
main()
{
    int a, b, sum ;
    a = 12 ;
    b = 34 ;
    sum = a + b ;
    printf("sum is %d\n", sum) ;
}
```

本程序的功能是求两整数之和。/*…*/部分是注释，是为了便于阅读程序而安排的，对程序的编译和执行没有影响。注释可以加在程序的任何位置。注释可以是任意一串字符，但不能再含有/*…*/，也就是说，注释是不能嵌套的(也没有必要)。这里的注释是用汉字给出的，你的计算机可能没装汉字，可用英文或汉语拼音来写注释。本书中采用汉字注释完全是为了阅读的方便。

程序中的

```
int a, b, sum ;
```

是变量定义部分，这里我们用 int 定义变量 a、b 和 sum 为整型。C 语言规定使用任何变量之前都要先定义。

```
a = 12 ;
```

```
    b = 34 ;
    sum = a + b ;
```

是三个赋值语句，“=”是赋值运算符，它的作用是把其右部表达式的值赋给左部的变量，这里是把值 12 赋给变量 a，把值 34 赋给变量 b，把 a+b 的结果赋给变量 sum。

```
    printf("sum is %d \n",sum);
```

中的"sum is"和"\n"我们已经熟悉，%d 是一个输出转换说明，意思是：输出时用一个整型值来代替它，这里，这个整型值是变量 sum 的值。于是本程序的输出结果为：

```
    sum is 46
```

例 1.3

```
    main()/*求两个数中较大者*/
    {
        int a, b, c ;
        printf("a,b=") ;
        scanf("%d,%d", &a, &b) ;
        c = max(a, b) ;
    printf("max=%d\n", c) ;
    }
    int max(a, b)  /*返回 x、y 中较大者*/
    int x, y ;
    {
        int  z;
        if ( x > y )
                z = x;
        else
                z = y ;
        return(z) ;
    }
```

本程序定义了两个函数：main 和被调函数 max。在 main 中，语句

```
    printf("a,b=") ;
```

输出

```
    a,b=
```

然后光标停在=之后。下一句

```
    scanf("%d,%d",&a, &b) ;
```

用于接收两个整型数据到变量 a 和 b 中。scanf 是 c 输入输出库中提供的输入函数，它要求按照转换字符(这里是%d，表示整型)将相应类型的数据输入到指定变量的存储单元中去，&a 表示取变量 a 的地址。

语句

```
    c = max(a, b) ;
```

是将函数 max 的值赋给变量 c，其中 a 和 b 是参数(称为实在参数)，执行时将变量 a 和 b 的值传给被调函数 max。max 的作用是将 x、y 中较大的数送 z，并通过语句

return(z)；

将其作为函数值返回给主调函数。函数头

```
int max(x, y)
int x, y ;
```

表示 max 带有两个形式参数 x 和 y，在 max 被调用时，这两个参数的值由主调函数中对应的实在参数(本程序中是 a 和 b 的值)传过来，函数体中求出 x、y 中较大的数值送 z，并返回主调函数，语句

```
if( x > y)
    z = x;
else
    z = y;
```

是 C 中的一种判定控制语句结构，它根据 x 是否大于 y 来选择执行

```
z = x
```

或

```
z = y
```

本程序的执行情况如下：

```
a,b=4,6↓
max=6
```

例 1.4

```
#include "stdio.h"
main()
{
    int c ;
    while(( c = getchar() ) ! = EOF )
            putchar(c);
}
```

这个程序的功能是将输入的内容复制到输出，第一行

```
#include "stdio.h"
```

是 C 语言中的文件包含，stdio.h 是一个头文件，通过文件包含命令 #include，程序把 stdio.h 包含在自己所在的文件中。stdio.h 是关于标准输入输出的头文件，其中包括使用标准输入输出库函数的许多信息。在本程序中，由于使用 getchar 和 putchar 函数(实际上是宏，将在第九章中介绍)，因此需要用命令包含，将存有必要信息的 stdio.h 包含进来。符号 EOF 也在 stdio.h 中定义，实际中，它定义为

```
#define EOF -1
```

意思是在程序中遇到 EOF 就用 -1 代替。EOF 是文件结束标志，对于任意一个文件，系统会自动在其尾部加一个 EOF 标志。当文件是从键盘上输入时，在微机 DOS 操作系统下，打入

^ z(ctrl+z)后，系统自动为输入流加上一个 EOF(UNIX 系统下，可打入 ^ d)。函数 getchar 用于读一个字符，putchar 用于输出一个字符，语句 while 表示当括号内的表达式值为真时，反复地执行后面跟着的语句，因此

```
while((c=getcharc())! =EOF)
    putchar(c);
```

的意思是：当读入的字符不是 EOF 时，就输出它，直到读入 EOF，所以我们可以这样运行这个程序

```
abc1234  xyz789^ z(输入)
abc1234  xyz789  (输出)
```

在本章的例子中，我们让每个语句单独占据一行的位置，并且书写时作了适当的缩进，这些都不是 C 语言语法所要求的，而是为了阅读方便，如例 1.4 也可以写成

```
#include "stdio.h"
main(){int c;while((c=getcharc())! =EOF)putchar(c);}
```

其结果也是一样的。C 语言不是靠回车，而是靠分号来标识一个语句的结束，回车的功能与一个空格是相同的。虽然可以把程序写成上面那个样子，而且可以正确地执行，但实际中几乎没有人那么写，即使是很短的程序。计算机界有句名言：程序是给人读的。看到这个命题，你也许会奇怪，程序不是为了在计算机上运行的吗?确实如此，但不仅仅如此，你写一个程序以后可能要修改，别人也可能会阅读或修改你的程序，在没有完全读懂的情况下做修改，将会带来许多麻烦，即使不改，一个表达清楚的程序也会给人带来愉快。再者，连你自己都不能清楚地表达思路的话，还能指望计算机作什么呢?

为了清晰而正确地写程序，首先要使程序具有良好的结构，如使用将实现细节隐蔽得很好的函数，少用外部变量，不采用非结构化的控制结构等。同时，也要注意书写风格，如每个语句独立占一行，适当的缩进，尽量选择有实际意义的标识符，适当的加入注释等等，这些在我们今后的讨论中还会反复提到。总之，从一开始写程序就要培养好的风格。

第三节　C 程序上机运行

学习编程，上机是非常重要的。在计算机上编程并运行，从结果中了解程序的运行过程，进而了解程序的结构，比只看书要有效得多。另外，你可以对已有的程序做各种修改，进而了解语言更多的知识，也可以向别人学到很多东西。上机可以提高你编程的兴趣，也可以提高自信心，当看到自己编写的程序在计算机上正确地运行时，你会感到自己的创造力。

本节中，我们介绍 turbo　C 上机的简单步骤，如果你使用其他的 C，或希望对 turbo　C 有更多的了解，请参考相应的用户手册。

用 turbo C 上机的步骤：

先将 turbo C 软件安装到机器的硬盘上。

1. 调入 turbo　C

只要打入 tc，然后回车，屏幕顶部将显示出 turbo C 的命令菜单(见图 1-2)

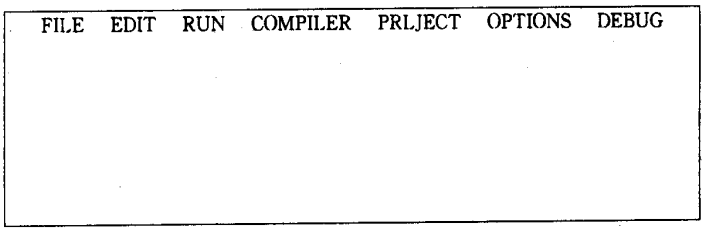


图 1-2 命令菜单

用键盘上的"←"和"→"键可以移动屏幕上的光标，如果光标不在顶部的命令菜单行中，可以打 F10 键将光标移到顶部。将光标移到"FILE"处，打回车键，FILE 下面出现一个子菜单，如图 1-3，用"↓"键将光标移到 Load(或 New)处打回车键，表示要输入源程序，屏幕上又出现一个小窗口，如图 1.4 所示，要求你输入文件名，此时可打入

file1.c ↵

如果原来没有这个名字的文件，将建立一个新文件，如果已有此文件，则将此文件调入并在屏幕上显示，然后自动转为编辑(EDIT)状态。

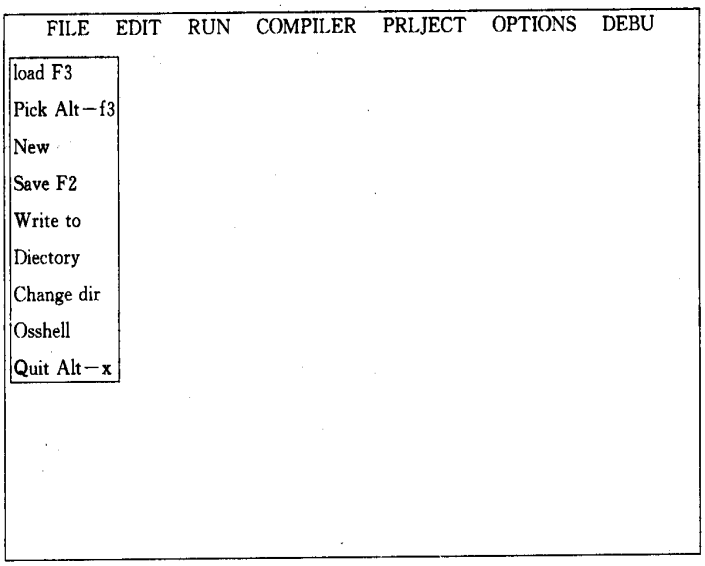


图 1-3 子菜单

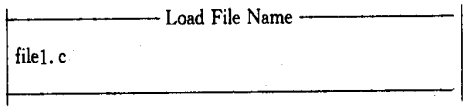


图 1-4 文件名输入框

你也可以直接进入编辑状态，这时在 DOS 命令行中打入

tc file1.c

系统就自动进入编辑屏幕。也可不打 c，这时系统取缺省值.c。

2. 编辑源文件。

你可以根据需要输入或修改源程序。

3. 编译源程序

只需要简单地打 F9 键，就可以进行源程序的编译和连接，并在屏幕上显示有无错误。如有错误，按任一键后，屏幕显示源程序，光标停在出错的位置上。屏幕的下半部分将显示出

有错误的行和错误原因。根据提供的信息可以修改错误，这时按下“F6”键后就可修改错误，修改完后再按“F6”键，接着打“↘”键，光标就停到下一个出错的行。所有错误修改完后，再按“F9”进行编译，如此反复，直到没有编译错误为止。

4. 执行程序

你只要打入 ^ F9(按下 ctrl 键的同时按 F9 键)即可执行程序。如果程序中需要输入数据，就在此时将所需数据输入，接着程序执行，输出结果。

如果发现结果不对，可再按步骤 2、3、4 修改程序、编译、执行程序，直到得到正确的结果。

5. 保存文件

编辑完源程序或准备退出 turbo C 时，也可打“F2”键将源程序存在硬盘上。

6. 退出 turbo C

同时按下 Alt 和 X 键，便退出 turbo C，回到操作系统命令状态。此时可以用系统命令来显示源程序或运行程序。如

type file1.c ↘(显示源程序清单)

file1 ↘(执行程序 file1.exe)

如果想再输入或修改源程序，可以再从步骤 1 开始。

以上的步骤是对一个程序只放在一个源文件中时采用的，如果一个程序放在多个文件中，就要使用 PROJECT 项生成所谓 prj 文件。考虑到初学时，程序通常都较小，不必放在多个文件中分别编译，这里就不多介绍了。

最后讲一下本书附带磁盘的使用方法。本书中全部例子的源程序附在一张磁盘上，文件名是字母 E 开头，后跟章、例号，如例 1.1 为

E1－1.c

你可以直接调入 turbo C 编译并执行这些例子，有些例子有多个程序，这时文件名中再含有一个标识版本的数字放在最后，如例 1.1 有两个版本的话，程序名分别是

E1－1－1.c

和

E1－1－2.c.

书中有些例子只给出了一个被调函数，而没有 main，为了便于上机操作，盘中都给了用于测试这些例子的主调函数，上机时也可以直接编译运行。

第二章　数据类型、运算符与表达式

本章学习重点

本章介绍C语言的基本数据类型：整型、浮点型、字符型，相应类型的变量和常量以及类型之间的转换，并介绍算术、赋值和逗号表达式的概念和使用。通过本章的学习，应能掌握C语言数据和运算的基本概念，为以后各章的学习打下基础。

第一节　C语言的数据类型

许多人总爱形象地把计算机程序与做菜用的菜谱做个类比。确实如此，菜谱中规定了厨师烹饪的步骤，就象计算机程序中一条条指令规定了计算机应怎样运行。菜谱中的指令总是把各种食品原料当作自己操作的对象，而计算机中要被处理的对象就是数据。

图2.1给出了C语言中所能处理的各种数据类型。

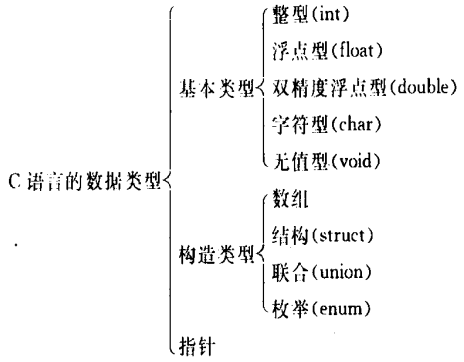


图2-1　数据类型

在本章中我们介绍基本数据类型，而其他数据类型留在后面章节中讨论。

第二节　标识符

标识符是一个字符序列，在C语言中用来标识常量、变量、函数、数据类型的名字。C语言规定：标识符只能由字母、数字及下划线"_"(不是减号)组成，并且数字不能做为标识符的第一个字符，下面是一些合法的标识符：

sum

a1

i

J5x3

Num_of_lines

iobuf

不合法的标识符如：

5t　　以数字开头

U.S　　出现了字符

Good bye　中间留有空格

C 语言规定，字母的大小写是有区别的，比如，SUM、Sum 和 sum 代表三个不同的标识符，如果你曾有过 PASCAL 或 FORTRAN 等语言的编程经历，那么一定要注意这一点。

C 语言本身并没有要求标识符的长度，但在任何机器上，所能识别的标识符的长度总是有限的，有些系统可以识别长达 31 个字符的标识符(如 VAX－11 VMS C)，而有些系统只能识别 8 个字符长度的标识符，这意味着：即使第 9 个字符不同，只要前 8 个字符一样，系统也认为是同一个标识符，如 category1 和 category2。因此，为了避免出错和增加可移植性，标识符最好前 8 个字符有所区别。

一个写得好的程序，特别是一个有实际意义的程序，标识符的选择应尽量反映出所代表对象的实际意思。如表示年可以用 year，表示长度可以用 length，表示和可以用 sum 等，这样的标识符本身就增加了可读性，使程序更加清晰。

C 语言中有一些标识符被称为关键字，它们是系统留做特殊用途的，不能作为一般标识符使用，如我们见过的整型变量说明 int 等。附录 B 中给出了 C 语言中全部关键字。

有些标识符虽不是关键字，但 C 语言总是以固定的形式用于专门的地方，因此，用户也不要把它们当做一般标识符使用，以免造成混乱。这些标识符常用的有：

define include ifdef ifndef endif elif

第三节　常量

常量是指在程序运行过程中其值不能被改变的量。

2.3.1　整型常量

整型常量简称整数，C 语言中有三种形式的整型常量。

1. 十进制整数

以我们通常习惯的十进制整数形式给出，如：

10，－345，0，＋5

2. 八进制整数

八进制整数是以 0 开头的八进制数字串，只能使用数字 0～7。八进制数没有符号，如果一个八进制前带有负号，实际上是对这个八进制数进行了单目减运算。下面是一些合法的八进制整数：

0123　　(十进 83)

0400　　(十进制 256)

0177777　　(十进制 65535)

3. 十六进制整数

十六进制整数是以 0x 或 0X 开头的十六进制字符串。十六进制字符有 0～9，a～f(或 A～F)，其中 a～f(或 A～F)表示十进制值 10～15，下面是一些合法的十六进制整数

0x12　　(十进制 18)

0xb	（十进制 11）
0xA8	（十进制 168）
0xFFFF	（十进制 65535）

整型数的取值范围通常是由机器的字长决定的。在一个字长 16 位的计算机上，带符号整数值的范围是十进制－32768～32767，无符号整数值的范围是十进制 0～65535。数值超过这个范围时，系统自动按长整数对待。长整数每个数值占 4 个字节(32 位)存储空间，长整数的表示形式是在数字的后面跟上字母 L，如

12L	（十进制 12）
－100L	（十进制－100）
1234567L	（十进制 1234567）
012L	（十进制 10）
0200000L	（十进制 65536）
0x12L	（十进制 18）
0x10000L	（十进制 65536）

看起来 12L 与 12 好象没有什么差别，但在一个用 16 位二进制表示整数的计算机上，12 占据 2 个字节存储空间，而 12L 占据 4 个字节。

2.3.2 浮点型常量

浮点型常量也称实数。C 语言中实数有两种表示形式。

1. 十进制数形式

由数字，小数点和可能的正负号组成，如

0.345，.345，345.，345.0，0.0，－3.45

2. 指数形式

也称科学计数法，用 e 或 E 表示指数，其一般形式为

aEb

表示 $a\times10^b$，其中 b 必须是整数，下面给出一些指数形式实数的例子：

345e2	（相当于 345×10^2）
－3.2e5	（相当于 -3.2×10^5）
.5e－2	（相当于 0.5×10^{-2}）

实数不管表现形式如何，总是占据 8 个字节的存储空间(即以双精度的形式出现)。

2.3.3 字符型常量

C 语言中的字符型常量是用单引号(向左撇)括起来的一个字符，如

'a','x','0','*','A',' '

C 语言的字符常量占据一个字节的存储空间，在那个存储单元中存放的实际上并不是字符本身，而是该字符在所在机器采用的字符集中的编码，也就是一个整数值。大多数机器系统采用 ASCⅡ字符集(本书中也假定是 ASCⅡ字符集，见附录 A)，在这种情况下，'a' 在内存中表示为对应的 ASCⅡ码 97，'0' 表示为 48，形式如图 2-2 所示。

图 2-2 字符常数的表示

由于字符常量是个整数，因此它可以象整数一样参加数值运算。当然，它的主要用途是与其他字符作比较。

除了以上形式的字符常量外，C 语言还允许使用一种以特殊形式出现的字符常量，以表示某些非图形字符，这就是以“\”开头的转义字符序列。在上一章中，我们用“\n”表示换行，\n 实际上是一个字符，它的 ASC Ⅱ 码值为 10，因此可以表示为'\n'。常见的以\开头的转义序列见表 2.1

表 2.1　转义字符序列表

字符形式	功能
\n	换行
\t	水平制表(下一个 tab 键的位置)
\v	垂直制表(竖向跳格)
\b	退格
\r	回车
\f	走纸
\0	空字符
\	反斜线
\'	单引号'
\ "	双引号"
\ddd	1～3 位 8 进制所代表的字符

\ddd 是用 1～3 位八进制数来表示相应的 ASC Ⅱ 码，如'\n'也可以表示为'\012'或'\12'。

2.3.4　字符串常量

字符串常量是用一对双引号(")括起来的零个或多个字符的序列，如

"This is a character string."

"CHINA"

"0123456789"

"$10000.00"

"a"

" "（引号中有一个空格）

""（引号中什么也没有）

"\n"(引号中有一个转义字符)

字符串常量在内存中存储时，系统自动为每个字符串常量的尾部加一个字符\0，用以标识这字符串的结束，如字符串

“CHINA”

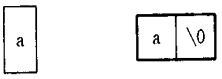
在内存中的形式是

图 2-3　字符常数的表示

(实际上，字母应当用对应的 ASC Ⅱ 码表示，但为了方便，以后表示字符时，直接用字符

本身表示)。

了解了这一点,你就可以区分字符串"a"和字符 'a' 有何不同了。它们在内存中的形式如图 2-4 所示



(a)'a'的存储表示　(b)"a" 的存储表示

图 2-4

字符串以\0 结束这种形式表明,C 中的字符串的长度是不受限制,其长度可以靠\0 来判断。

2.2.5 符号常量

常量也可用一个标识符来命名,这就是符号常量,如

例 2.1

```
#define PI 3.1415926
main()
{
    float radius,circum,area;
    printf("%d",&radius) ;
    circum = 2 * PI * radius ;
    area = PI * radius * radius ;
    printf("circumference is%f\ n", circum);
    printf("area is%f\n", area) ;
}
```

运行结果

```
Input radius:3
circumference is 18.849556
area is 28.274334
```

这个程序是按输入的半径值,求圆周及圆的面积。第一行的

```
#define PI 3.1415926
```

定义了一个符号常量 PI,以后凡在程序中出现 PI,都表示 3.1415926。PI 是一个常量,在程序中只能引用,而不能被改变。用符号常量来代替常数本身,至少有两个好处:

其一,可以使程序更清晰易读,如我们定义了一个表示每页行数的符号常量

```
#define PAGESIZE 55
```

则语句

```
while (line <= PAGESIZE)
    line ++ ;
```

比较清楚地表现了当行(line)小于每页尺寸时应做的工作。如果用

```
while l(line<=55)
    line ++ ;
```

就很难弄清楚 55 代表什么。

其二，程序更易修改。当我们把页的大小改变为 66 行时，只需要改写

#define PAGESIZE 66

而程序的其余部分不用改变，如果我们不用符号常量而用数值本身，就需要在程序中到处寻找数值 55，而且并不是所有 55 都表示是每页的尺寸，这就大大增加了修改的难度，也很容易出错。

通常的习惯是用大写字母表示符号常量，用小写字母表示变量等，以示区别。

最后提醒一点，定义符号常量的 define 行不要以分号结束，它不是一个语句。

第四节　变量及其说明

变量是指程序执行过程中，其值可以改变的量。

变量有一个名字，称为变量名，用标识符表示。每个变量都与一个数据类型相联系，类型决定了变量可以取值的范围和它可以参加的运算。C 语言规定：

所有变量在使用之前必须加以说明

变量的说明形式为

类型名　变量名，变量名，……变量名；

2.4.1　整型变量

整型变量用关键字 int 说明。

如

```
int i, j, k ;
```

说明了三个整型变量 i，j 和 k。

整型变量的取值范围由机器字长决定，如微机的整型变量占据二个字节，取值在－32768～32767(－2^{15}～2^{15}－1)之间，而整型变量占 4 个字节的机器(如 VAX－11)，整型变量取值范围是－2147483648～2147483647(－2^{31}～2^{31}-1)。

除了基本 int 型外，还可以在 int 前加上修饰符来改变 int 型的意义，修饰符有：

signed(有符号)

unsigned(无符号)

long(长型)

short(短型)

由于整型的缺省形式是有符号的，所以 signed 可以不用(确实极少看到有人用)，加上修饰符后，整型变量的形式有：

short int　　可简写为 short

long int　　可简写为 long

unsighed int　　可简写为 unsigned

unsigned short int　可简写为 unsigned short

unsigned long int　可简写为 unsigned long

也就是说，整型变量的说明加上修饰符后，int 可以省略。整型变量加上修饰符后，其取值范围有所变化。以 16 位机为例，表 2.2 给出了各种形式整型变量的取值范围。

表 2.2　整型量取值范围

类型	所占位数	取值范围
int	16	-32768～32767
short	16	-32768～32767
long	32	-2147483648～2147483647
unsigned	16	0～65535
unsigned short	16	0～65535
unsigned long	32	0～429497295

如果是 32 位机，则 int 和 unsigned 的取值与 long 和 unsigned long 相同。

例 2.2

```
main()
{
    int a, b, c, d ;
    a = - 1 ;
    b = 4 ;
    c = 6 ;
    d = (a + b) * c ;
    printf("%d\n", d) ;
}
```

运行结果

18

2.4.2　浮点型变量

浮点型变量也称实型变量，有单精度和双精度之分。

单精度浮点型变量用关键字 float 说明，双精度浮点型变量用关键字 double 说明。如：

```
float f, g ;
double d ;
```

一般来讲，float 型变量占据 4 个字节(32 位)存储空间，绝对值取值范围约在 10^{-38}～10^{38}之间，取 6～7 位有效数字；double 型占据 8 个字节(64 位)存储空间，取值范围由具体机器决定，有 12～16 位有效数字，除 float 和 double 型外，ANSI C 还增加了 long double 类型，占 16 字节(128 位)，有效数字约 24 位，但多数 C 编译中没有设置。

前面的例 2.1 就是一个使用浮点型变量的例子。

2.4.3　字符型变量

字符型变量由关键字 char 说明，

如

```
char c1, c2 ;
```

一个字符型变量占据一个字节(8 位)存储空间，只能存放一个字符。不要以为它能够存放一个字符串，如：

```
c1 = 'a' ;
```

是正确的，而

```
c2 = "a" ;
```

则是错误的。

字符型变量的值实质上是一个 8 位的整数值，因此取值范围一般是－128～127，char 型变量也可以加修饰符 unsigned，则 unsigned char 型变量的取值范围是 0～255（有些机器把 char 型当做 unsighed char 型对待，取值范围总是 0～255）。

C 语言把字符型量当做一个较小的整型量，可以象整型量一样使用它。

例 2.3

```
main()
{
    char c1 = 'a' ;
    char c2 = 'b' ;
    char c3,c4;
    c3 = c1 - ('a' - 'A') ;
    c4 = c2 - ('a' - 'A') ;
    printf("%c%c\n", c3, c4) ;
}
```

运行结果为

```
A  B
```

程序中的 'a' － 'A' 是大小写字母之间的差值（别忘了字符对应相应的 ASC Ⅱ 码），其值为 32。也可以把程序写成

```
c3 = c1 - 32 ;
c4 = c2 - 32 ;
```

效果是一样的，printf 中的%c 表示输出一个字符。

字符型数据也可以用整数形式输出，如将输出语句改为

```
printf("%d%d\n", c3, c4) ;
```

则运行结果为

```
65  66
```

2.4.4 变量赋初值

变量定义之后，我们可以用赋值的形式给它确定的值，也可以在定义变量的同时给它设置初值。如例 2.3 也可写成

```
main()
{
    char  c1 = 'a' ;
    char  c2 = 'b' ;
    char  c3,  c4;
    c3 = c1 - ('a' - 'A') ;
    c4 = c2 - ('a' - 'A') ;
```

```
    printf("%c%c\n", c3, c4);
}
```

在这里

```
char c1 = 'a';
```

就相当于

```
char c1;
c1 = 'a';
```

两条语句。

第五节　算术运算符和算术表达式

2.5.1　算术运行符

C 语言中有 5 种算术运算符

+　(加法运算符)

−　(减法运算符)

*　(乘法运算符)

\　(除法运算符)

%　(模运算符)

这里的+、*、\、%都是双目运算符,即它们在参与运算时,左右各需要一个运算对象(运算数),而"−"既是双目运算符,又是单目运行符。当作为单目运行符时,只需要后跟一个运算对象,表示取它的负值。

符号"*"表示乘,在 C 语言中(其他语言也大致如此)不能用数学中习惯的×或·号表示乘,也不能什么都不写,如:2 * a 不能写做　2a,2×a 或 2·a。

符号/表示除,不能用÷号(键盘上没有它)。当除数和被除数都是整数时,其商也是整数,如 5 / 3 结果为 1。

符号%只能用于整数,表示求模运算,即两个数相除的余数,如 5 % 3 结果为 2,9 % 4 结果为 1。

2.5.2　算术表达式

C 算术表达式由运算对象(常量、变量、函数等)、圆括号和运算符组成。最简单的情况,一个常量,一个变量(赋过值的)都是合法的表达式,如 5,0,x 等。作为一般情况,则可有更多的运算符和圆括号,如

```
-a / (b1 + 0.5) - 11 % 7 * 'a'
```

要注意,C 表达式中的所有字符都是写在一行上的,没有分式,也没有上下标,括号只有圆括号一种(方括号和花括号作其他用途),如数学表达式

$$\frac{a+b}{c+d}$$

需要写成

```
(a+b)/(c+d)
```

这里括号是不可缺少的,如没有括号,实际上就变成了

$a+\frac{b}{c}+d$

算术运算符的优先级是单目减最高，然后是乘除，最后是加减，圆括号可用来改变优先级。图 2-5 给出 C 语言中算术运算的优先级与结合规则。

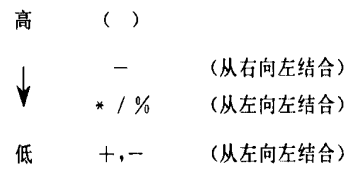


图 2-5　运算符的优先级和结合性

了解了算术运算符的优先级和结合规则，我们可以看一下表达式－a / ((b1 + 0.5) － 11 % 7 * 'a' 的求值过程：

①求－a 的值，

②求 b1+0.5 的值，

③求①/②的值，

④求 11%7 的值，

⑤求④ * 'a' 的值，

⑥求③－⑤的值。

C 表达式不论简单还是复杂，最终总是可以求出值，也就是说，C 表达式的本质是一个值。因此，表达式可以出现在数值能够出现的任何地方，这也意味着，如果表达式中有变量，则变量在被引用之前，必须是已赋过值的。

例 2.4　求二次方程的根，假定 b^2-4ac≥0。

```
#include "math.h"
main()
{
    float a, b, c, x1, x2 ;
    printf("Input a,b,c(b * b－4 * a * c>=0)\n") ;
    scanf("%f%f%f", &a, &b, &c) ;
    x1 = (－b + sqrt(b * b － 4 * a * c) ) / (2 * a) ;
    x2 = (－b － sqrt(b * b － 4 * a * c) ) / (2 * a) ;
    printf("x1=%f,x2=%f\n", x1, x2) ;
}
```

运行结果

```
Input a,b,c(b * b－4 * a * c>=0)
1  －6  5
x1=5.000000,x2=1.000000
```

程序中求平方根的库函数 sqrt 需要用到文件 math.h 中的信息，所以在程序最前面加上 include 命令。要注意的是，求 x1 和 x2 时表达式中的括号都是必须的，如果写成

－b+sqrt(b * b－4 * a * c)/(2 * a)

实际上相当于数学表达式

$$-b+\frac{\sqrt{b^2-4ac}}{2a}$$

而写成

```
(-b+sqrt(b*b-4*a*c)/2*a
```

则相当于数学表达式

$$\frac{-b+\sqrt{b^2-4}}{2}\cdot a$$

显然这都是不对的。当把一个用分式表示的数学表达式转为C表达式时，一个较好的方法是分子、分母分别用括号括起。

2.5.3 类型转换

如果一个运算符两边的运算数类型不同，先要将其转换为相同的类型，即较低类型转换为较高类型，然后再参加运算，转换规则如图2-6所示。

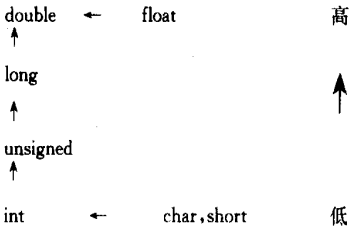


图2-6 类型转换

图中横向箭头表示必须的转换，如两个float型数参加运算，虽然它们类型相同，但仍要先转成double型再进行运算，结果亦为double型。纵向箭头表示当运算符两边的运算数为不同类型时的转换，如一个long型数据与一个int型数据一起运算，需要先将int型数据转换为long型，然后两者再进行运算，结果为long型。所有这些转换都是由系统自动进行的，使用时你只需从中了解结果的类型即可。

上面说的转换是自动的，而另一方面，C语言也提供了以显式的形式强制转换类型的机制，其一般式为

(类型名)(表达式)

其效果是把表达式的类型强制转换为要求的类型，而不管类型的高低，如

```
(double)a        (将a转换成double型)
(int)(x+y)       (将x+y的结果转换为int型)
```

要转换的表达式要用括号括起，如

```
(int)(x+y)
```

与

```
(int)x+y
```

是不同的，后者相当于(int)(x)+y，也就是说，只将x转换成整型，然后与y相加。

在许多场合，强迫转换是必须的，如用sqrt求一个整数n的平方根，sqrt要求一个双精度型参数，如果用整型则可能得不到正确的结果，因此必须强制转换n为double型，写成

```
sqrt((double)n)
```

(有些系统可以自动将整型量转换为double型再求平方根，但你不应该依赖系统，因为那会

降低程序的可移植性)。

当较低类型的数据转换为较高类型时,一般只是形式上有所改变,而不影响数据的实质内容,而较高类型的数据转换为较低类型时则可能有些数据丢失。

例 2.5

```
main()
{
int  i, j ;
float  x, y ;
i = 1 ;
x = 2.5 ;
j = (int)x ;
y = (float)i ;
printf("i=%d,y=%f\n", i, y) ;
printf("x=%f,j=%d\n", x, j) ;
}
```

运行结果为

```
i=1,y=1.000000
x=2.500000,j=2
```

这里 y 是由 i 转换而得到的,其值与 i 只是形式上不同,而 j 是由 x 转换得到的,由于是从转高的 float 型转换为较低的 int 型,这时,小数点后面的内容被省略掉了。

第六节　增和减运算符

C 语言对增加和减少变量的值提供了两个独特的运算符:

单目运算符++表示对变量加 1

单目运算符--表示对变量减 1

++和--既可用作前缀也可用作后缀,如

```
++i (先把 i 值加 1,然后再引用)
i++ (先引用 i 值,然后把 i 值加 1)
```

在有些场合,前缀和后缀效果是一样的,如

```
while (line <= PAGESIZE)
    line ++ ;
```

和

```
while(line<=PAGESIZE)
    ++line ;
```

得到相同的结果。在这种情况下,你可以凭自己的喜好采用任何一种方式,而在另外一些场合,效果却是不同的。

例 2.6

```
main()
{
    int i ,j, k, l ;
    i = j = 3 ;
    k = i ++ ;
    l = ++ j ;
    printf("i=%d,j=%d,k=%d,l=%d\n", i, j, k, l) ;
}
```

开始时 i、j 都被赋初值 3,语句

```
k = i ++ ;
```

先把此时的 i 值 3 赋给 k,使 k 的值为 3,然后 i 自己加 1,值变为 4。而语句

```
l = ++ j ;
```

先是 j 自身加 1,值变为 4,然后把值 4 赋给变量 l,l 的值也是 4,于是输出的结果为

```
i=4,j=4,k=3,l=4
```

需要注意的是:

增 1 减 1 运算都要求运算对象是变量

因此

```
++ 5
(x + y) --
```

都是错误的。

++和--运算符有和单目减一样的优先级,也就是说,比 * 、/,%都要高,它们同样是按从右到左结合的。

例 2.7

```
main()
{
    int  i = 2 ;
    printf("%d\n", -i++) ;
    printf("%d\n", i) ;
}
```

输出结果为

```
-2
3
```

程序中的表达式

```
-i++
```

相当于

```
-(i++)
```

也就是先取 i 的值 2,求负后输出-2,然后 i 自身加 1,输出 3。

需要提醒注意的是

增1减1操作有可能带来副作用

例2.8

```
main()
{
    int  i = 3 ;
    printf("%d,%d,%d\", i, i ++, i ++) ;
}
```

先考虑一下,结果应该是多少,是3,4,5吗?在多数C中,printf中各输出参数的求值是从右向左的,也就是先求最后一个i++的值,得到3后i自增1,再求前一个i++,得到值4后i再自增1,最后求最左边的i值,变成了5,所以输出是

5,4,3

如果你希望输出3、4、5的话,可把输出语句改写成

```
printf("%d,%d,%d\n", i, i + 1, i + 2) ;
```

结果将是

3,4,5

这里你可以看出

i ++

与

i + 1

并不是一回事儿。

第七节　赋值运算符和赋值表达式

2.7.1　赋值运算符

C语言中的赋值运算符是"=",它的功能是把其右侧表达式的值赋给左侧的变量,赋值的一般形式为

变量=表达式

如

a=1

表示把数值1赋给变量a,也就是让变量a具有数值1。

赋值运算符"="不表示其左右两侧的内容相等(C语言中判断相等用符号"=="),而表示把表达式的值送到变量代表的存储单元中去。由此,赋值运算符的左侧只能是变量,因为它表示一个存放值的地方,象

```
1=a;
a+b=c;
```

这样的赋值显然是不合法的。

2.7.2　赋值中的类型转换

当赋值运算符两边的运算对象类型不同时,将要发生类型转换,转换的规则是:

把赋值运算符右侧表达式的类型转换为左侧变量的类型

具体的转换如下。

1. 浮点型与整型

将浮点数(单双精度)转换为整数时,将舍弃浮点数的小数部分,只保留整数部分。

将整型值赋给浮点型变量,数值不变,只将形式改为浮点形式,即小数点后带若干个0。

我们把例2.5稍做修改,去掉强制转换操作,把语句

```
j=(int)x;
y=(float)i;
```

改成

```
j=x;
y=i;
```

其余不变,运行结果仍是

```
i=1,y=1.000000
x=2.500000,j=2
```

由此可以看出,赋值时的类型转换实际上是强制的。

2. 单、双精度浮点型

由于C语言中的浮点值总是用双精度表示的,所以float型数据只是在尾部加0延长为double型数据参加运算,然后直接赋值。

double型数据转换为float型时,通过截尾数来实现,截断前要进行四舍五入操作。

3. char型与int型

int型数值赋给char型变量时,只保留其最低8位,高位部分舍弃。

char型数值赋给int型变量时,一些编译程序不管其值大小都作正数处理,而另一些编译程序在转换时,若char型数据值大于127,就作为负数处理。对于使用者来讲,如果原来char型数据取正值,转换后仍为正值;如果原来char型值可正可负,则转换后也仍然保持原值,只是数据的内部表示形式有所不同。

4. int型与long型

long型数据赋给int型变量时,将低16位值送给int型变量,而将高16位截断舍弃。(这里假定int型占两个字节)。

将int型数据送给long型变量时,其外部值保持不变,而内部形式有所改变。

5. 无符号整数

将一个unsigned型数据赋给一个占据同样长度存储单元的整型变量时(如:unsigned→int,unsigned long→long,unsigned short→short),原值照赋,内部的存储方式不变,但外部值却可能改变。

将一个非unsigned整型数据赋给长度相同的unsigned型变量时,内部存储形式不变,但外部表示时总是无符号的。

例2.9

```
main()
{
    unsigned a, b ;
```

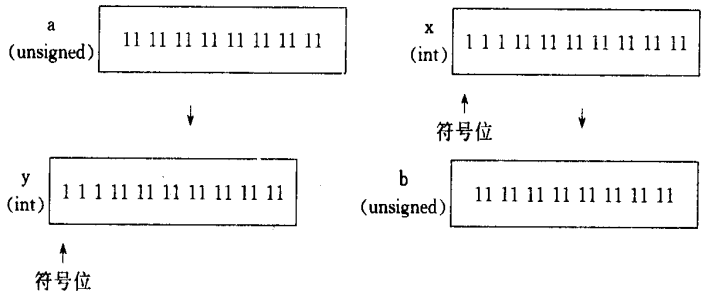
```
    int i, j ;
    a = 65535 ;
    i = -1 ;
    j = a ;
    b = i ;
    printf("(unsigned)%u→(int)%d\n", a, j) ;
    printf("(int)%d→(unsigned)%u\n", i, b);
}
```

运行结果为

(unsigned)65535→(int)−1

(int)−1→(unsigned)65535

这里%u 用来输出一个无符号数，图 2.7 给出了转换的过程。



(a)无符号数转为有符号数　　(b)有符号数转为无符号数

图 2-7

计算机中数据用补码表示，int 型量最高位是符号位，为 1 时表示负值，为 0 时表示正值。如果一个无符号数的值小于 32768 则最高位为 0，赋给 int 型变量后，得到正值。如果无符号数大于等于 32768，则最高位为 1，赋给整型变量后就得到一个负整数值。反之，当一个负整数赋给 unsigned 型变量时，得到的无符号值是一个大于 32768 的值。

C 语言这种赋值时的类型转换形式可能会使人感到不精密和不严格，因为不管表达式的值怎样，系统都自动将其转为赋值运算符左部变量的类型，而转变后数据可能有所不同，在不加注意时就可能带来错误。这确实是个缺点，也遭到许多人们批评。但不应忘记的是：C 语言最初是为了替代汇编语言而设计的，所以类型变换比较随意。当然，用强制类型转换是一个好习惯，这样，至少从程序上可以看出想干什么。

2.7.3 复合型赋值运算符

形如

i = i + 2

的表达式，其意思是把 i 的内容取出加工，然后再赋给 i，这时可用一个复合的赋值式运算符写成如下形式

i += 2

这样表示更紧凑，也易于理解，我们可以把它解释为“加 2 到 i”。当变量名较长时，我们不必把注意力放在关注两边变量否一致上。

复合赋值运算符的形式是：

变量 OP＝表达式

其含义是

变量＝变量 OP(表达式)

这里 OP 是

＋,－,＊,/,%,〈〈,〉〉,&,^ ,|

之一(后 5 个在介绍位运算时再介绍),如

x /= 2

相当于

x = x / 2

要注意表达式两边的括号,如

a *= b + 3

实际上是

a = a * (b + 3)

而不是

a = a * b + 3

2.7.4 赋值表达式

赋值运算符连接变量和表达式而得到的式子就是赋值表达式,如

a = 2

也就是说,这种形式不仅可以做为一个语句出现,也可以出现在表达式可以出现的任何地方(实际上赋值语句在 C 语言中被认为是一种表达式语句),既然是表达式,也就具有一个值。赋值表达式的求值过程是:先求赋值运算符右部表达式的值,然后把这个值赋给左部的变量,而赋值表达式的值就是这时左部变量的值。

赋值运算符(包括前面介绍的＋＝,－＝等)的优先级很低,仅高于逗号运算符,其结合性是由右向左。

在 C 程序中,经常可以看到形如

i = j = k = 0

的式子,它相当于

i = (j = (k = 0))

效果是对变量 i,j,k 赋予了相同的值 0。赋值表达式也经常出现在条件或循环语句的判断条件中,这种用法在后面各章随处可以见到。

第八节　逗号运算符和逗号表达式

逗号运算符就是我们常用的逗号“,”,作为操作符时,它可以把多个表达式连接起来,如

a + 5 , b - 3

就是一个逗号表达式。

逗号表达式的求值过程是从左到右,逐个求表达式的值,最后整个表达式的值取最右一个表达式的值。

例 2.10

```
main()
{
    int  i, j ;
    i = 20 ;
    j = (i -- = 4, i / 2) ;
    printf("i=%d,j=%d\n", i, j) ;
}
```

运行结果为

i=16,j=8

我们来分析一下程序的执行,语句

```
j = (i -- = 4, i / 2) ;
```

的执行顺序是,首先求 i -- = 4,也就是 i = i - 4,求得 i 值为 16,然后求 i / 2,结果为 8,最后把 8 作为逗号表达式的值赋给变量 j。注意:语句中的括号是必须的,因为逗号运算符是所有 C 运算符中优先级最低的一个,如果不用括号,它实际上相当于

```
(j = i -- = 4), i / 2
```

实际中,逗号表达式只在 for 语句中才经常被用到,其他地方基本上不用,我们这里的例子只是为了说明它的用法。

第九节　常见错误

这一节里,我们列出一些学习本章内容编程时初学者容易犯的错误,以供参考。

1. 忘记定义变量

如:

```
main()
{
    x = 1 ;
    y = 2 ;
    printf("%d\n", x + y) ;
}
```

这个程序编译时系统会指出错误:变量 x 和 y 没有定义。C 语言规定,所有变量在使用之前都必须定义。正确的格式应在语句

```
x = 1 ;
```

之前加上说明

```
int x, y ;
```

2. 变量没有赋值就引用。

如:

```
main()
{
    int x, y, z ;
```

```
    z = x + y ;
    printf("%d\n", z) ;
}
```

这个程序在编译时会给出警告，告诉你变量x、y没有赋值就使用了。如果要执行这个程序，输出将是一个混乱的值，在程序中变量应该先赋值后再引用。

3. 整型变量赋值超过整型变量的取值范围

如

```
main()
{
    int i, j, k ;
    i = j = 100 ;
    k = 10 * i * j ;
    printf("%d\n", k) ;
}
```

这个程序在编译时不会产生任何问题，但如果在PC机上运行，却得不到预期的结果，程序不会输出值100000，而是输出－31072，这是因为int型变量只能取值于－32768～32767之间，而100000已起出了这个范围，高位被截断，因而得到了这个莫名其妙的结果。如果你需要计算较大整数值时，可把变量定义为long型，输出时用%ld进行输出，如把程序写成

```
main()
{
    int i,j ;
    long k ;
    i = j = 100 ;
    k = 10 * i * j ;
    printf("%ld\n",k);
}
```

这里的输入结果就为100000。

4. 将数学表达式改变C算术表达式时，丢失必要的括号

如有数学表达式

$$\frac{a+b}{c \cdot d}$$

写成了

```
a + b / c * d
```

这实际上相当于

$$a+\frac{b}{c} \cdot d$$

显然不对，正确的写法应是

```
(a + b) / (c * d)
```

5. 向表达式赋值

有些初学者搞不清赋值运算符的含义，总是按数学中的习惯，认为“=”表示相等，于是就有可能写出

```
0 = x ;
```

这样的语句。要记住：赋值表达式的左部只能是变量。

6. 用 scanf 输入数据时忘记地址运算符

如：

```
int a, b ;
scanf("%d%d", a, b) ;
```

这种形式的输入，在有些编译中会给出警告，但程序仍可执行，只是不能得到正确的输入值，C 语言要求输入参数应是变量的地址值。所以，输入应写成

```
scanf("%d%d", &a, &b) ;
```

7，在 scanf 中加入"\n"

许多初学者受 printf 影响总是把输入写成

```
scanf("%d\n", &a) ;
```

实际上这不是一个错误，但在执行时，输入数据并打入回车后，程序仍不继续运行，使人莫名其妙，再次打回车后程序才会运行，这是因为在 scanf 的控制字符串中，所有非格式转换字符和非空格字符在输入时都需要一个相同的字符作为匹配，这样你就要多打入一个回车符，虽然不能算是真正的错误，却带来许多麻烦。

8. 语句末尾忘记分号

C 语言的语句以分号结尾，如果不写分号，在编译时，将会指出错误，而且这个错误并不是告诉你少写了分号。

9. 括号不匹配

在写表达式时，经常是少写了右边的括号，这时编译也会指出错误，但可能不会告诉你什么地方少了括号，而是显示其他类型的错误。

10. 向字符变量赋字符串值

如

```
char c ;
c = "abc" ;
```

字符型变量只能存放一个字符，而不能存放字符串。

第三章　逻辑运算和判定结构

本章学习重点

本章介绍C语言的关系运算、逻辑运算以及用作判定控制的if语句和switch语句。学习本章应能掌握关系和逻辑运算的概念，并能够用if和switch语句编制具有判定控制结构的程序。

判定控制结构是结构程序设计所采用的三种基本控制结构之一。另外两种是顺序控制和循环控制。有人曾经证明：任何程序都可用顺序、判定、循环三种控制结构来实现。而结构化程序设计的研究成果表明：只用这三种控制结构编写的程序易于保证正确性。在编制程序时，有时并不能保证程序一定执行某些指令，而是要根据一定的外部条件来判断哪些指令要执行。如菜谱中要加工西红柿，可能有这样的步骤：如果是用鲜西红柿，则去皮，切碎，开始时放入，如果是用西红柿酱，就在最后放入。这里，我们并不知具体操作时执行哪段指令，但菜谱给出了不同条件下的处理方式。在计算机程序中也是如此，可以根据不同的条件执行不同的代码段。这就是判定控制结构。

第一节　关系运算符和关系表达式

3.1.1　关系运算符

C语言提供一组关系运算符，用以表示两个运算对象之间的大小关系。表3.1给出了这些关系运算符。

表3.1　关系运算符

运算符	含义
<	小于
<=	小于或等于
>	大于
>=	大于或等于
==	等于
! =	不等于

关系运算符都是双目运算符，它们的优先级比算术运算符低，高于赋值运算符，结合规则是从左到右。在关系运算符中，<、<=、>、>=同级，它们高于==和！=，==和！=同级。

3.1.2　关系表达式

用关系运算符将两个表达式连接起来的式子就是关系表达式。下面给出一些合法的关系表达式。

a > b, 8 < 5, a + b <= c + d, (i = j + k) ! = 0

关系表达式的值是一个逻辑值，即非真即假的值。于是，表达式

8 < 5

的值为假，而表达式

a > b

的值将取决于 a 与 b 的具体值，但只可能是真或假两种情况之一。

C 语言中没有专门的逻辑型数据，而是用 0。表示假，用 1(或非 0)表示真。

因此，若 a=3,b=2，则 a > b 的值为 1，而 8 < 5 的值为 0。

因为关系运算符的优先级低于算术运算符，所以

a + b <= c + d

实际上相当于

(a + b) <= (c + d)

也就是判定 a+b 的和是否小于等于 c+d 的和。虽然不是功能上的需要，但加上括号可以使程序更清楚易读，所以许多程序员习惯在关系表达式中用括号括起的算术表达式。

关系运算符的优先级高于赋值运算符，所以

(i = j + k) ! = 0

中的括号有没有是不一样的。上面表达式表示把 j+k 的和赋给 i，然后判定 i 是否不等于 0，如果去掉括号，实际上相当于

i = (j + k) ! = 0

也就是先判断 j+k 的和是否不为 0。然后把逻辑值 0 或 1 赋给 i。

第二节　逻辑运算符和逻辑表达式

3.2.1　逻辑运算符

逻辑运算符用于逻辑运算，也就是真假值的运算。C 语言提供三种逻辑运算符：

&&　逻辑与

||　逻辑或

!　逻辑非

“&&”和“||”是双目运算符，它们需要两个运算量，如

(a > b) && (c > d)

(x == 0) || (y == 0)

“!”是单目运算符，只需要后跟一个运算量，如

! (a > b)

逻辑运算符要求所操作的运算量是逻辑量(数值量也当做逻辑量对待)，结果也是逻辑值，如

(a > b) && (c > d)

的值是这样确定的，如果 a>b 为真，并且 c>d 为真，则(a>b)&&(c>d)为真，否则为假。表 3-2 给出了逻辑运算中当 a 与 b 的值为不同组合时，逻辑运算所取的值，此表称为真值表。

表 3.2　逻辑运算真值表

a	b	! a	! b	a&&b	a||b
真	真	假	假	真	真
真	假	假	真	假	真
假	真	真	假	假	真
假	假	真	真	假	假

逻辑运算符的优先级是:逻辑非! 最高,逻辑与 && 次之,逻辑或||最低,它们与算术运算符、关系运算符、赋值运算符等的关系如图 3-1 所示。

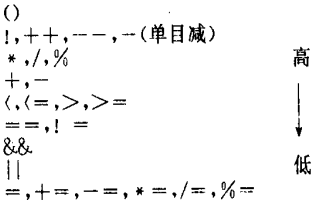


图 3-1　部分运算符的优生级

由图 3-1 的优先级可以看出

a > b && c > d　　相当于(a > b) && (c > d)

x == 0 || y == 0 相当于(x == 0) || (y == 0)

! a && b == c　　相当于(! a) && (b == c)

你可能会觉得 C 语言运算符的优先级多而且复杂,其实,这还只是其中的一部分。如果你要怕记不住它们的关系,可以按照自己的实际意思使用括号"()"。实际中,许多聪明的程序员总是用圆括号来表明运算的先后,即使有时候圆括号并不影响运算顺序。这样的做的好处是程序不容易出错,阅读起来也方便。

逻辑运算符的结合规则为:&& 和||左结合,! 右结合。

3.2.2　逻辑表达式

逻辑表达式用逻辑运算符连接运算量,并产生一个逻辑值。C 语言中没有专门的逻辑值,真假值用 0、1 代替。实际上,当判断一个值为非 0 时,认为逻辑值为真,一个值为 0 时,逻辑值为假。而在产生逻辑值时用 1 表示真,用 0 表示假。

如果 a=3,b=2 则

! a　相当于! 3 为假,值是 0

a && b　为真,值是 1

a && ! b　相当于 3&&0,值为 0

! a || b　为真,值是 1

运算符 && 和||具有这样的性质:它们从左到右计算各运算量的值,一旦能够确定表达式的值就不再继续运算下去,如

a && b

如果 a 为假(0),就不必再求 b 的值了,因为逻辑与运算只有当运算符两侧的运算量都为真(非 0)时才取真值,而 a 为假已决定了表达式的值为假。如果 a 为真,则要继续求 b 的值,以判断表达式的真假。与此类似,

a || b

一旦 a 为真，也就不再求 b 的值了。由于这个性质，有时运算量的先后顺序是很重要的，如

x ! = 0 && y / x < 1

先判断 x 是否为 0，只有当 x 不等于 0 时才求 y/x，并判断是否小于 1，而 x 一旦为 0，就不再求 y/x，从而避免了除数为 0 的情况。

第三节 if 语句

3.3.1 if 语句的简单形式

if 语句用来判定给定的条件是否满足，根据结果（真或假）来选择执行相应的操作。它的最简单的形式为

if(表达式)

语句

其含义是：如果表达式为真（非 0），则执行其后所跟的语句。要注意，这里没有 then，因为省略 then 后含义仍是很清楚的，如果你有 PASCAL、FORTRAN 等语言的编程经验，则要小心，不要画蛇添足。

例 3.1

```
/ * 求一个整数的绝对值 * /
main()
{
    int n ;
    printf("Input a number:") ;
    scanf("%d", &n) ;
    if (n < 0)
        n = -n ;
    printf("The absolute value is %d\n", n) ;
}
```

运行结果

```
Input a number:-5
The absolute value is 5
```

再次运行

```
Input a number:10
The absolute value is 10
```

这种形式 if 语句的执行过程是这样的：首先计算 if 后面表达式的值，如果它为真（非 0），就执行后面跟着的语句；如果为假（0），就跳过整个 if 语句，其逻辑结构如图 3-2 所示。

图 3-2 中表达式后的语句表示是一条语句，如果需要多条语句，就要用花括号将它们括起来，形成一个分程序结构，称为复合语句，它等效于一条语句。

例 3.2

```
/ * 按由小到大顺序输出 3 个整数 * /
```

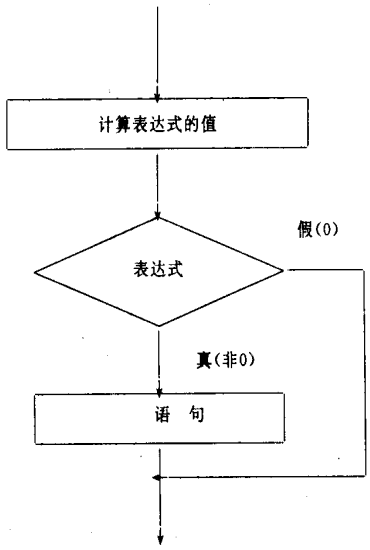


图3-2 if语句逻辑结构

```
main()
{
    int a, b, c, temp ;
    printf("Input three numbers\n") ;
    scanf("%d%d%d", &a, &b, &c) ;
    if (a > b) {
        temp = a ;
        a = b ;
        b = temp ;
    }/*这里不需要分号*/
    if (a > c) {
        temp = a ;
        a = c ;
        c = temp ;
    }
    if (b > c) {
        temp = b ;
        b = c ;
        c = temp ;
    }
    printf("%d,%d,%d\n", a, b, c) ;
}
```

运行情况

Input three numbers

5 8 2

2 5 8

现在我们来看一下程序段

```
if (a > b) {
    temp = a ;
    a = b ;
    b = temp ;
}
```

如果a、b的大小与要求的从小到大的顺序不符，就要把它们交换一下，这时需要一个临时变量，使用三条语句完成互换，而if语句的控制范围只限一条语句，这就要用花括号将三个语句括起，组成一个复合语句。请注意，复合语句的后面不需加分号，C语言中只有复合语句的后面不需要分号，因为花括号可以明确地表明语句的结束(而且一个函数的函数体就相当于一个复合语句，在函数的最后加个分号，看着总有点不太舒服)。

当然，我们也可以用逗号运算符将三个语句连接成一个语句，写成

```
if (a > b)
    temp = a , a = b , b = temp ;
```

这时if后只有一个语句，可以不用花括号。但大多数程序员不习惯这样表示，所以还是用花括号为好。

3.3.2 if－else结构

if语句更一般的形式是

```
if(表达式)
    语句1
else
    语句2
```

其含义是：如果表达式为真(非0)，就执行语句1，否则执行语句2，这里的语句1和语句2可以是一条语句，也可以是用花括号括起的复合语句。if－else形式的逻辑结构如图3-3所示。

例3.3

```
/*输入一个整数，判断它是奇数还是偶数*/
main()
{
    int n ;
    printf("Input a number\n") ;
    scanf("%d", &n) ;
    if (n % 2 == 0)
        printf("The number is even\n") ;
    else
        printf("The number is odd\n") ;
}
```

运行结果

```
Input a number
```

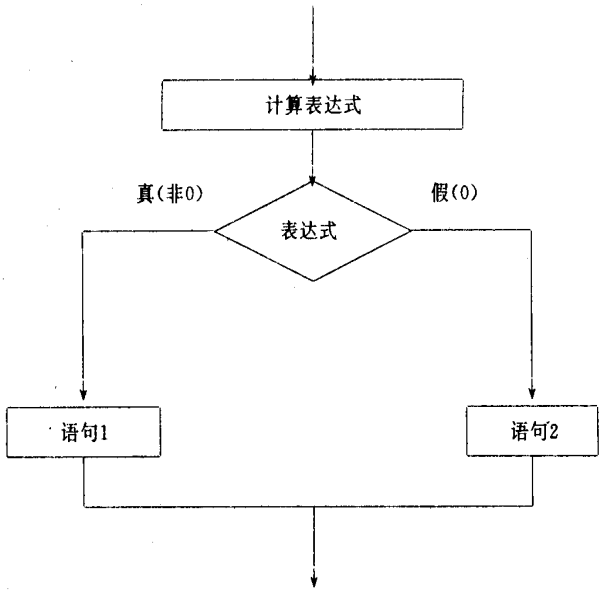


图3-3 if-else结构

100

The number is even

再次运行

Input a number

99

The number is odd

如果你曾经是个 PASCAL 程序员,就要注意,else 之前的语句后的分号不能丢,因为 PASCAL 中分号用于分隔两个语句,而 C 中的分号是标识一个语句的结束。

下面,我们再来看一个例子。这个例子中,if 语句的条件表达式稍微复杂一些。

例 3.4 判断某一年是否闰年。

闰年是其年份能够被 4 整除,但不能被 100 整除的年,或者是能被 400 整除的年。因此闰年的条件可用逻辑表达式表示为

(year % 4 == 0 && year % 100 ! = 0) || (year % 400 == 0)

式中的括号不是必须的,但可以把较长的表达式分成两部分,这样看起来清楚一些。完整的程序如下:

```
/ * 判断某年是否为闰年 * /
main()
{
    int year ;
    printf("Type in a year\n") ;
    scanf("%d", &year) ;
    if ( (year % 4 == 0 && year % 100 ! = 0) || (year % 400 == 0) )
        printf("%d is leap year\n", year) ;
    else
```

```
        printf("%d is not a leap year\n", year);
}
```

运行一次

```
Type in a year
1993
1993 is not a leap year
```

再次运行

```
Type in a year
2000
2000 is a leap year
```

再运行一次

```
Type in a year
1900
1900 is not a leap year
```

3.3.3 else-if 结构

if—else 结构可对只有两种可能的条件做判断，而实际中有些问题可能需要在多种情况中做判断，如数学中的符号函数，它定义为：

$$\text{sign}=\begin{cases}1 & x>0\\ 0 & x=0\\ -1 & x<0\end{cases}$$

这种情况用前面介绍的 if--else 结构就显得不得心应手了。幸好 if 语句允许嵌套，也就是说 if 语句中还可以包括另一个 if 语句，而 if 语句嵌套的最常见的形式是 else—if 结构，它可以很方便地解决这类问题。

else—if 结构的一般形式为：

```
if(表达式 1)
    语句 1
else if(表达式 2)
    语句 2
    ⋮
else if(表达式 n)
    语句 n
else
    语句 n+1
```

其含义是：如果表达式 1 为真，则执行语句 1，否则，如果表达式 2 为真，则执行语句 2，…，依此类推。如果表达式 n 为真，则执行语句 n，如果各表达式都不为真，则执行语句 n+1。其逻辑结构由图 3-4 给出。

现在我们就用这种结构来编一个求解符号函数的程序。

例 3.5

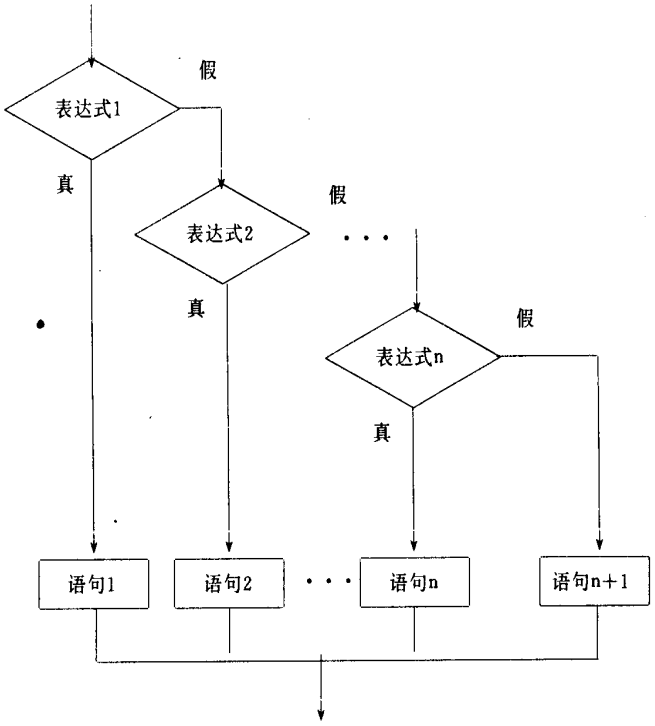


图3-4 else-if结构

```
/*求解符号函数*/
main()
{
    int x, sign ;
    printf("please input a number\n") ;
    scanf("%d", &x) ;
    if (x > 0)
        sign = 1 ;
    else if (x == 0)
        sign = 0 ;
    else
        sign = -1 ;
    printf("The sign is %d\n", sign) ;
}
```

运行一次

```
please input a number
-100
The sign is -1
```

再次运行

```
please input a number
2
```

The sign is 1

再次运行

please input a number

0

The sign is 0

3.3.4 if 语句的嵌套

if 语句中还可以包含 if 语句，这在前面的 else－if 结构中已经看到。但并不是所有的嵌套都能表示成 else－if 结构，更一般的情况是 if 后和 else 后的语句都可以再包含 if 语句。

例 3.6　求一个点所在的象限。

```
main()
{
    float x, y ;
    printf("Input the coordinate of a point\n") ;
    printf("x=") ;
    scanf("%f", &x) ;
    printf("y=") ;
    scanf("%f", &y) ;
    if(x > 0)
        if (y > 0)
            printf("The point is in 1st quadrant. \n") ;
        else
            printf("The point is in 4th quadrant\n") ;
    else
        if (y > 0)
            printf("The point is in 2nd quadrant\n") ;
        else
            printf("The point is in 3rd quadrant\n") ;
}
```

运行情况

Input the coordinate of a point

x=5

y=3

The point is in 1st quadrant.

再运行一次

Input the coordinate of a point

x=－2

y=－7

The point is in 3rd quadrant.

这个程序没有考虑点在 x 轴或 y 轴时的情况。

if 语句中的 else 并不总是必须的，在嵌套的 if 结构中，可能有的 if 语句带有 else，有的 if 语句不带 else，那么一个 else 究竟与哪个 if 配对呢？C 语言规定

else 总是与前面最近的 if 相配对。

例如

```
if (n > 0)
    if (a > b)
        c = a ;
    else
        c = b ;
```

这里，else 与内层的 if 结合，如果你希望 else 与外层的 if 相结合的话，你不能写成

```
if (n > 0)
    if (a > b)
        c = a ;
else
    c = b ;
```

因为缩进只是为了便于阅读，计算机执行时是不理睬的，它总是把 else 与前面最近的 if 相结合。如果要想使 else 与前面的 if 配对，办法就是用花括号，如

```
if (n > 0) {
    if (a > b)
        c = a ;
}
else
    c = b ;
```

这样 else 就与外层的 if 相配对了。

3.3.5 条件运算符

求 a、b 中较大者可以用语句

```
if (a > b)
    max = a ;
else
    max = b ;
```

来实现，类似这种 if 和 else 各带一个赋值语句的结构，C 语言提供了一种专门的条件运算符，上面语句可以用条件运算符实现。

```
max = (a > b) ? a : b ;
```

其中(a>b)? a:b 是一个条件表达式，执行过程为：如果 a>b 为真，则表达式取 a 的值，否则取 b 的值。

运算符“?”是 C 语言中唯一的一个三目运算符，它连接三个运算量，一般形式为

表达式 1? 表达式 2:表达式 3

条件表达式的逻辑结构如图 3-5 所示。

条件运算符的优先级较低，只高于赋值运算符和逗号运算符，而低于算术运算符、关系

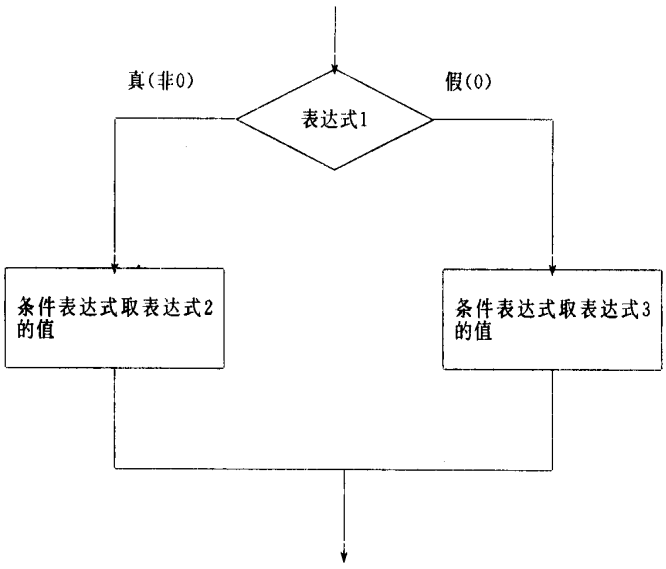


图3-5　条件表达式的逻辑结构

运算符和逻辑运算符，如

(a > b) ? a − b : b − a

相当于

(a > b) ? (a − b) : (b − a)

条件运算符采用右结合，但在实际中几乎看不到一个表达式中出现两次条件运算符。

条件表达式的本质也是一个值，因此它也可以出现在值可以出现的任何地方。如语句

printf("%d\n", a > b ? a : b);

输出 a、b 中较大者。

条件表达式体现了 C 语言简洁明快的风格，这种紧凑的表现形式是 C 语言区别于其他一些高级语言的一个显著特点。

第四节　switch 语句

在前面我们曾说过：else−if 结构提供了一种多分支选择的功能，这种结构的问题在实际中是经常遇到的，为此，C 语言专门提供了一种用于多分支选择的语句：switch 语句，它可以处理 else−if 结构的问题，而且更清楚。switch 语句的一般形式如下：

```
switch(表达式){
case 常量 1:
语句序列 1
case 常量 2:
语句序列 2
    ⋮
case 常量 n:
语句序列 n
```

```
default:
语句序列 n+1
}
```

其含义是:先计算表达式的值,如果值与哪个常量相匹配,就执行哪个 case 后的语句序列;如果表达式的值与所有列举的常量都不相同,则执行 default 后的语句序列。由于 case 及 default 后都允许是语句序列,所以,当你要安排多个语句时,也不必用花括号括起。

switch 语句中的 default 项不是必须的,如果没有 default,则所有的常量都不与表达式的值匹配时,switch 语句就不执行任何操作。另外,default 写成最后一项也不是语法上必须的,它也可写在某个 case 前面,但习惯上人们总是倾向把 default 写在最后。

例 3.7　输入形如 3+5,9/3 的四则运算式,求其计算的结果。

```
/*计算表达式的值*/
main()
{
    float a, b ;
    char op ;
    printf("Enter your expression\n") ;
    scanf("%f%c%f", &a, &op, &b) ;
    switch (op) {
    case '+' :
        printf("result is %0.2f\n", a + b) ;
        /*%0.2f 表示输出的实数保留二位小数*/
        break ;
    case '-' :
        printf("result is %0.2f\n", a - b) ;
        break ;
    case '*' :
        printf("result is %0.2f\n", a * b) ;
        break ;
    case '/' :
        if (b == 0)
            printf("Division by zero\n") ;
        else
            printf("result is %0.2f\n", a / b) ;
        break ;
    default :
        printf("Unknown operator\n") ;
    }
}
```

运行结果

```
Enter your expression
56.5+18.7
result is 75.20
```

再次运行

```
Enter your expression
5/0
Division by zero
```

再运行一次

```
Enter your expression
3.5#4.2
Unknown operator
```

你可能已经注意到了，在每个 case 后的语句序列中，最后一个语句都是 break，设置它是为马上结束 switch 语句，如果不写 break 语句，则还要继续执行下面所有 case 与 default 内的语句，例如，去掉例 3.7 中的 break，并输入

```
4 + 2
```

则输出为

```
result is 6.00
result is 2.00
result is 2.00
result is 8.00
Unknown operator
```

显然与实际要求的不相符。

在实际中有时会遇到几种情况要求执行相同的操作，这时用 switch 语句写程序时，只要在共同的语句序列之前列出多个常量即可。例如，我们可以把例 3.7 改一下，使它在接收乘法表达式时也能识别形如 5X6 形式的表达式. 下面是程序中相应的改动部分

```
switch (op) {
    ⋮
case '*' :
case 'X' :
    printf("result is %0.2f\n", a * b) ;
    break ;
    ⋮
}
```

这里 * 和 X 执行相同的操作。

第五节　常见错误

1. 忘记必要的逻辑运算符。

如

```
if (a > b > c)
⋮
```

本意为如果a>b并且b>c。由于数学中使用a>b>c的形式，也就把它搬到计算机程序中。而在C中，a>b>c的求值是先求a>b，得到一个逻辑值0或1，再拿这个数与c作比较，结果当然是不对的。对于这种情况，应使用逻辑表达式，写成

```
if (a > b && b > c)
⋮
```

2. 误把“=”作为等于运算符。

如

```
if (x = 1)
⋮
```

本义是如果x等于1，而x=1并不是关系表达式，是一个赋值表达式，这时表达式的值永远为真(非0值1)，而不管x原来是什么值。正确的等于运算符是==，由于受数学或其他语言的影响，这种错误是经常出现的。上面的式子应写成

```
if (x == 1)
```

或者写成

```
if (1 == x)
```

这样，当忘记一个=而写成

```
if (1 = x)
```

时编译程序就会为你指出错误。

3. 该用复合语句时忘记写花括号。

如

```
if (a > b)
    temp = a ;
    a = b ;
    b = temp ;
```

由于没有写花括号，if的影响只限于

```
temp = a ;
```

一条语句，而不管

```
(a > b)
```

是否为真，都将执行后两个赋值，正确的写法应为

```
if (a > b) {
    temp = a ;
    a = b ;
    b = temp ;
}
```

4. 在不该加分号的地方加分号。

如

```
if (a == b) ;
    c = a + b ;
```

本义是如果 a 等于 b 则执行 c=a+b,但由于 if(a+b)后跟有分号,c=a+b 在任何情况下都要执行。因为 if 后加分号相当于后跟一个空语句(第四章中将介绍空语句),这种错误是因为习惯在每行的后尾都加分号所致。正确的写法应是

```
if (a == b)
    c = a + b ;
```

5. else 之前的语句丢失分号。

如

```
if (a > b)
    max = a
else
    max = b;
```

这种错误来自 PASCAL 编程的习惯。在 PASCAL 中 ELSE 之前一定不能出现分号,因为 PASCAL 中的分号是语句的分隔符,不可能有语句以 ELSE 开头,而在 C 中,分号是语句的结束符,一个语句的末尾必须要有一个分号,正确的写法是

```
if ( a > b)
    max = a ;
else
    max = b ;
```

6. 在 switch 语句中忘掉了必要的 break,如

如

```
switch (score) {
    case 5 : printf("very good") ;
    case 4 : printf("good") ;
    case 3 : printf("pass") ;
    case 2 : printf("fail") ;
    default: printf("error") ;
}
```

当 score 是 5 时,输出为

very good good pass fail error

原因是丢失了 break 语句。正确的写法应是

```
switch (score) {
    case 5 :
        printf("very good") ;;
        break ;
    case 4 :
        printf("good") ;
        break ;
```

```
    case 3 :
        printf("pass") ;
        break ;
    case 2 :
        printf("fail") ;
        break ;
    default:
        printf("error") ;
}
```

第四章　循环控制结构

本章学习重点

本章介绍C语言中用于循环控制的语句：while语句，do－while语句和for语句，以及通常是与它们配合使用的空语句，break语句，continue语句和goto语句。学习本章应掌握各种语句的用法并能用它们编写循环结构的程序。

许多实际问题中都需要有规律地重复某些操作，如菜谱中可以有："打鸡蛋直到泡沫状"这样的步骤，也就是说，在鸡蛋没有打成泡沫状时反复地打。相应的操作在计算机程序中就体现为某些语句的重复执行，这就是所谓循环。循环控制结构也是结构化程序设计的三种基本控制结构之一。为了方便地处理循环问题，C语言提供了三种用于循环控制的语句：while、for和do-while语句。

第一节　while循环

4.1.1　while语句

while语句的一般形式为

while(表达式)
语句

while语句的执行过程是：首先计算表达式的值，当值为真(非0)时执行其后跟的语句，每执行完一次语句后，再次判断表达式的真假，为真时继续执行语句，为假时结束while语句。也就是说，当表达式为真时，反复执行其后跟的语句。这里的语句部分称为循环体，它可以是一条单独的语句，也可以是复合语句。while语句的逻辑结构见图4.1给出。

例4.1　求 $\sum_{i=1}^{10} i$

```
/*用while语句求 Σ(i=1,10) i */
main()
{
    int i, sigma ;
    sigma = 0 ;
    i = 1 ;
    while (i <= 10) {
        sigma = sigma + i ;
        printf("i=%2d,sigma=%2d\n", i, sigma) ;
```

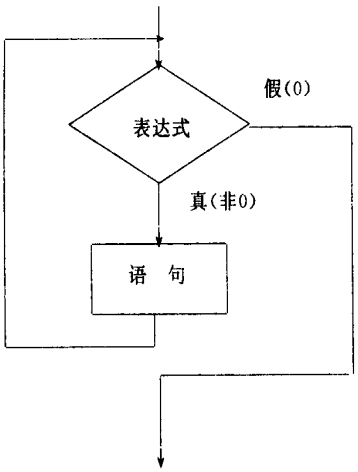


图4-1 while语句的逻辑结构

```
        i ++ ;
    }
}
```

运行结果为

```
i= 1,sigma= 1
i= 2,sigma= 3
i= 3,sigma= 6
i= 4,sigma=10
i= 5,sigma=15
i= 6,sigma=21
i= 7,sigma=28
i= 8,sigma=36
i= 9,sigma=45
i=10,sigma=55
```

从运行结果中我们可以看出程序的执行过程：在 while 语句执行前 sigma=0,i=1,执行 while 语句时，首先判断 i<=10 是否为真，由于 i=1，小于 10，因此开始执行循，sigma 变成 1，输出

```
i=1,sigma=1
```

然后 i 增加 1，变成 2，执行完循环体后，再次判断条件 i<=10，此时 i 等于 2，条件仍为真，继续执行循环体，sigma 在原来的基础上加上 i 值 2，结果为 3，printf 语句输出

```
i=2,sigma=3
```

循环体中 i 再增加 1，然后继续 while 条件的判断，依此类推，循环共运行 10 次，第 10 次时 i ++使 i 变成 11，这时再次判断 while 循环条件 i<=10，值为假，循环结束。

程序中，while 语句的循环体由多个语句组成，这时要用花括号将它们括起，构成复合语句，如果不写花括号则变成

```
while (i <= 10)
```

```
        sigma = sigma + i ;
        printf("i=%2d,sigma=%2d\n", i, sigma) ;
        i++ ;
```

while 语句的控制范围只限于

```
    sigma = sigma + i ;
```

一条语句，虽然写成缩进形式，计算机也不认为后两条语句是循环体中内容。由于循环中 i 并没有改变，所以条件总是真，于是循环体被一直执行下去，导致死循环。

例 4.2　求两个正整数的最大公因子。

我们采用 Euclid(欧几里德)算法来求最大公因子，其算法是：

1°　输入两个正整数 m 和 n。

2°　用 m 除以 n，余数为 r，如果 r 等于 0，则 n 是最大公因子，算法结束，否则 3°。

3°　把 n 赋给 m，把 r 赋给 n，转 2°。

例如：有 m=49，n=21，用 m 除以 n，余数为 7，由于 7≠0，将 n 赋给 m，r 赋给 n，这时 m=21，n=7，再次用 m 除以 n，21 除以 7，余数为 0，此时的 n 值 7 就是 49 和 21 的最大公因子。

程序如下

```
/* 用 Euclid 算法求两个正整数的最大公因子 */
main()
{
    int m, n,r ;
    printf("please type in two positive integers\n") ;
    scanf("%d%d", &m,&n) ;
    while (n ! = 0) {
        r = m % n ;/* 求余数 */
        m = n ;
        n = r ;
    }
    printf("Their greatest common divisor is %d\n", m) ;
}
```

运行结果

```
please type in two positive integers
49 21
Their greatest common divisor is 7
```

再运行一次

```
Please type in two positive integers
50 100
Their greatest common divisor is 50
```

在这个程序中判别的是 n 是否为 0，这是因为，在循环体中 r 的值已赋给了 n，因此判断的就是原来 m%n 的余数，printf 中输出的 m 也是如此，这是原来的 n 值，也就是最大公因

子。最后说一下

```
while (n ! = 0)
```

也可以写成

```
while (n)
```

因为 n 非 0 时为真，与 n! =0 为真是一致的，当然你可能觉得 n! =0 更清楚些。

4.1.2 空语句

上面两个例子中，循环体都要做具体操作。在实际中你还可能遇到循环体什么都不做的情况，这时相当于循环体是一个空语句。

C 语言中设置了一种空语句，也就是什么也不做，什么也不写，只占一个语句位置的语句。你可能还记得：C 语句以分号结束，一个空语句也以分号结束。

下面来看一个例子。

例 4.3 越过输入中的空白符，输出第一个非空白字符。

C 语言中把空格、制表符(\t)和换行符(\n)称为空白符，程序如下：

```
/ * 输出第一个非空白字符 * /
#include "stdio. h"
main()
{
    char c ;
    while ( (c = getchar() ) == ' '|| c == '\t' || c == '\n')
        ;
    putchar(c) ;
}
```

运行结果

```
    nonspace     (输入)
n                (输出)
```

程序中 while 语句的条件部分比较复杂，我们来分析一下。首先看

```
c = getchar()
```

它是一个赋值表达式，getchar 函数返的一个字符送到 c 中，然后

```
(c = getchar() ) ==' '
```

判断赋值表达式的值，也就是 c 这个字符是否为空格。由于赋值运算符的优先级低于关系运算符，所以括号是不可省略的。同样可以继续判断 c 是否为制表符或回车符，三个判断用 || 运算符连接，判断输入的字符是否为空白符。程序的要求是跳过所有空白符，因此当遇到空白符时只需要继续读入字符和判断，而这些操作都在 while 的条件部分完成，这样，实际上就不需要循环体中再做什么操作了，所以循环体中只用了空语句，语句

```
while ( (c = getchar() ) = ' ' || c == '\t' || c == '\n')
    ;
```

也可以写成

```
c = getchar() ;
while (c == ' ' || c == '\t' || c == '\n'))
```

```
    c = getchar() ;
```

这时就没有空语句了，但在实际中，许多程序员都倾向于把程序写得紧凑，他们更愿意把程序写得象前者，所以在C程序中经常可以看到空语句。

把

```
while ( (c = getchar() ) == ' ' || c == '\t' || c == '\n')
        ;
```

写成

```
while ( (c = getchar() ) == ' ' || c == '\t' || c == '\n');
```

也是可以的，但为了清晰起见，空语句也单独占一行为好。

最后作为对照，我们给出不用空语句求解例4.3问题的完整程序。

例4.4

```
#inclnde "stdio.h"
main()
{
    char c ;
    c = getchar() ;
    while (c == ' ' || c == '\t' || c == '\n')
        c = getchar() ;
    putchar(c) ;
}
```

第二节　for语句

在许多情况下，循环条件的变化是有规律的，我们希望把循环条件初值，判别条件，和循环条件的改变放在一起，这样便于控制，看起来也更清楚些。for语句就提供了这种机能。

例4.5　用for语句求 $\sum_{i=1}^{10} i$

```
main()
{
    int i, sigma ;
    sigma = 0 ;
    for (i = 1; i <= 10; i ++) {
        sigma = sigma + i ;
        printf("i=%2d,sigma=%2d\n", i, sigma) ;
    }
}
```

运行结果与例4.1完全一样。

通过与例4.1比较你可以发现，与循环条件有关的i的变化都放在了for语句中，过程控制更集中，也更容易阅读和理解。

for 语句的一般形式为

for(表达式 1;表达式 2;表达式 3)

语句

其执行过程是：

1° 先计算表达式 1 的值。

2° 计算表达式 2，如果值为真(非 0)，则执行其后跟的语句，否则 for 语句结束。

3° 计算表达式 3，转 2°。

for 语句的逻辑结构如图 4-2 所示。

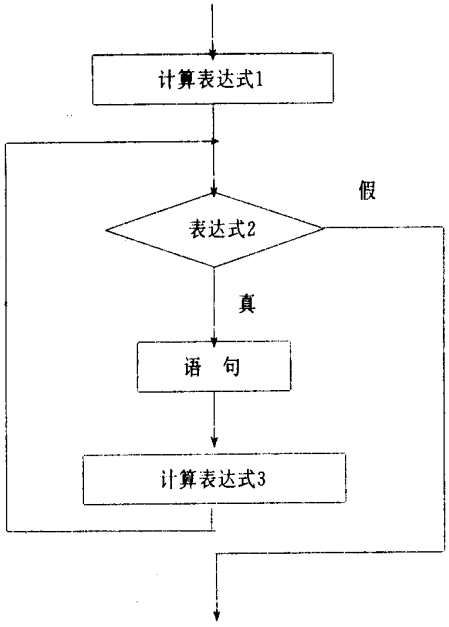


图4-2 for语句的逻辑结构

实际上 for 语句等价于如下形式的 while 结构

```
表达式 1 ;
while (表达式 2) {
    语句
    表达式 3 ;
}
```

比较一下例 4.1 和例 4.5 你就可以体会到两者之间的关系了。例 4.5 体现了 for 语句最普遍使用，也是最容易理解的形式：

for(循环变量初值;循环终止条件;循环变量增值)

语句

让我们再来看两个这种形式的例子。

例 4.6 输入 10 个字符，输出最大的 ASC Ⅱ 值

```
/ * 输出最大的 ASC Ⅱ 码值 * /
#include "stdio. h"
#define NUM 10
```

```
main()
{
    int i, c, max ;
    max = 0 ;
    for (i = 1; i <= NUM; i ++)
        if ( (c = getchar() ) > max)
            max = c ;
    printf("The largest ASCII value is %d\n", max) ;
}
```

运行结果

```
12 AB9 xab
The largest ASCII value is 120
```

此程序首先让 max 等于 0,然后每次读入字符后就与 max 比较,当 max 比输入字符的 ASCII 值小时,就把较大的 ASCII 值赋给 max,这样 max 在循环过程中总保存有到此时为止最大的 ASCII 码值,最后当循环结束时,输出的值就是 10 个字符中最大的 ASCII 码值。

前面两个例子中,循环变量都是从 1 开始取值,这不是必须的,这是问题本身的要求,下面的例子,循环变量不从 1 开始。

例 4.7 求菲波那契数列的前 20 个数

菲波那契(Fabonacci)序列起源于中世纪的意大利,问题是这样给出的:假定每对兔子每个月生出新的一对兔子来,新的每对兔子过两个月就可以生育。其次,这些兔子都不死,这样第一个月有一对兔子,第二个月有两对兔子,第三个月有 3 对兔子(第一个月的一对兔子又生了一对),第四个月有 5 对兔子(第二个月已有的两对兔子又各生了一对),依次类推,问一年共有多少对兔子。

把这个问题抽象一下可以看出,一般每个月的兔子数为上个月已有的兔子(因为兔子不死)和上上月已有兔子新生的兔子(兔子两个月后可以生育)之和,也就是说,序列中的某一项等于前两项之和,当然,一开始时不成立。

于是得到了所谓菲波那契序列,它定义为

$$F_n=\begin{cases}0 & n=1\\ 1 & n=2\\ F_{n-1}+F_{n-2} & n>2\end{cases}$$

也就是从第三项起每项都是前两项的和。

程序如下:

```
/* 求 Fabonacci 数列中的前 20 个数 */
main()
{
    int i, a, b, c ;
    a = 0 ;
    b = 1 ;
    printf("%10d%10d", a, b) ;/* 输出头两个 Fabonacci 数 */
```

```
    for (i = 3; i <= 20; i ++) {/*从第三项起循环*/
        c = a + b ;/*前两项之和*/
        printf("%10d", c) ;/*输出*/
        if (i % 5 == 0)/*每行输出5个值*/
            printf("\n") ;
        a = b ;/*项移动*/
        b = c ;
    }
    printf("\n") ;
}
```

运行结果为

```
    0         1         1         2         3
    5         8        13        21        34
   55        89       144       233       377
  610       987      1597      2584      4181
```

程序中用变量c表示F_n，用a和b表示F_{n-2}和F_{n-1}，菲波那契数列在求得一次F_n后，下一次时F_{n-1}应变成新的F_{n-2}，F_n应变成新的F_{n-1}，所以循环的最后用语句

```
a = b ;
b = c ;
```

表示这种移动。语句

```
if (i % 5 == 0)
    printf("\n") ;
```

表示每输出5个值换一行。因为i的值从3开始，因此执行循环三次时输出第一个换行符，此时循环输出的三个值和循环前输出的两个值加在一起，刚好是5个值，以后都是每循环5次输出一个换行符。如果for循环从1开始到i<=18，每行输出5个值的操作就不那么容易写了，从这里也可以看出，循环初值并不非得从1开始。

从以上的例子中可以看到，在执行for语句时，通常是先求循环初值，然后判断表达式2，如表达式2为真，就执行循环体，然后再执行表达式3，表达式3的执行通常要影响表达式2的条件(如i++改变了i值，而表达式2中用到i)，以便在循环一定次数之后，因表达式2的条件不能满足而结束循环。如表达式2的值始终不改变，循环就会一直继续下去，这是我们不希望看到的(也可以从循环体中强制退出循环而不判断表达式2，后面要讲)。

在for语句中，表达式1、表达式2、表达式3可以省略其中任一个、两个，甚至可以三个都不写。但即使是这时，用于分隔表达式的分号也不能省略。

实际上，当省略了表达式1和表达式3时，for语句就相当于while语句。如我们求$\sum_{i=1}^{10} i$也可以用如下的程序段来完成。

```
i = 1 ;
```

```
for ( ; i <= 10; ){
    sigma = sigma + i ;
    i ++ ;
}
```

这里省略了表达式 1 和表达式 3，其中 for 行就相当于

```
while (i <= 10)
```

当表达式 2 被省略时，程序无法靠 for 语句本身的条件来控制结束，这时要求循环体中一定要有能转出循环的语句，如

```
for (i = 1; ; i ++) {
    sigma = sigma + i ;
    if (sigma > 1000)
        break ;
}
```

表示当 sigma 值超过 1000 时结束循环。语句 break 在这里的作用是跳出循环。这段程序中由于循环结束的条件是判断 sigma 而不是 i，就是把控制部分写到 for 语句头中，与循环初值、循环增量部分的关系也不明显，倒不如在循环值中用条件语句判断反而还清楚些。

当 for 语句中的三个表达式都省略时，其形式为

```
for( ; ; )
```

它表示一种恒真的条件，在这种情况下，很显然退出循环要靠循环体中的语句来完成。能完成退出循环的语句有 break，goto，return 和函数 exit。

恒真条件也可用 while (1)表示，两者的功能完全相同，但喜欢用 for 语句的程序员似乎要多些，这大概是因为从形式上看 for (; ;)更醒目的缘故。

当循环条件恒真时，循环体中一定要有语句能跳出循环。

最后，我们来看一下在 for 语句中使用逗号表达式的情形。逗号表达式是用逗号运算符将多个表达式连接起来组成一个表达式。在 for 语句中，有时循环初始值的设置(表达式 1)和循环变量的改变(表达式 3)会用到逗号表达式(当然表达式 2 也可用逗号表达式，但实际上几乎没人这么用)，如求 $\sum_{i=1}^{10} i$，可用下面语句完成。

```
for (sigma = 0, i = 1; i <= 10; i ++)
    sigma = sigma + i ;
```

这里表达式 1 是一个逗号表达式，它设置了两个初值，类似的用法在以后还会出现。

第三节　do－while 语句

C 语言提供的第三种循环语句是 do－while 语句，其一般形式为

do
语句
while (表达式)

其执行过程是：先执行作为循环体的语句，然后判断表达式的值，如为真就继续执行循环体，

为假则结束循环，图 4-3 给出 do-while 语句的逻辑结构。

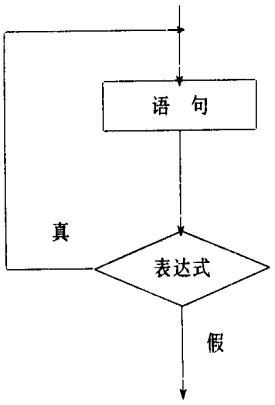


图4-3 do-while语句的逻辑结构

例 4.8

/* 用 do—while 语句求 $\sum_{i=1}^{10} i$ */

```
main()
{
    int i, sigma ;
    sigma = 0 ;
    i= 1 ;
    do {
        sigma = sigma + i ;
        printf("i=%2d,sigma=%2d\n", i, sigma) ;
        i ++ ;
    } while (i <= 10) ;
}
```

运行结果与例 4.1 相同。

do-while 语句与 while 语句的差别在于：while 语句先判断后执行，do-while 语句先执行后判断。当条件不满足时，while 语句不执行循环体中的内容，而 do—while 语句至少要执行一遍循环体。

你可以想像到 do—while 语句的使用要比 while 和 for 语句的使用少得多，但在有些场合 do—while 是必要的，至少也是方便的。

例 4.9 将一个整数的各位数字颠倒后输出。

对于这个问题，我们需要首先提取最后一个数字输出，这可用取模 10 的余数来求得，然后去掉最低位再取模 10 的余数就得到次低位，依次类推，可得到整数数字的反序。

```
/* 用 do—while 语句颠倒整数的各位 */
main()
{
    int i, r ;
```

```
    printf("Input an integer\n") ;
    scanf("%d", &i) ;
    do {
        r = i % 10 ;
        printf("%d", r) ;
    } while ( (i /= 10) ! = 0) ;
    printf("\n") ;
}
```

运行结果

```
Input an number
1234
4321
```

循环体中语句

```
r = i % 10 ;
```

求最后一位的余数,接着用 printf 输出它。在 while 部分,首先做

```
i /= 10
```

它相当于

```
i = i / 10
```

由于 i 是整数,整数相除的商仍是整数,运算的结果就是去掉 i 的最后一位数。这时再判断剩下的部分是否为 0,如为 0 则表明所有的位都已经转换完了,循环结束,如不为 0,就说明此整数中还有没有被转换的位数,就继续执行循环体。这里用 do—while 循环,不管输入的数据有多少位总要进行颠倒。这是很自然的,因为输入的数至少有一位,做一次转换是不会出错的,如

再次运行

```
Input an integer
0
0
```

与用 do—while 循环解决此问题相对照,下面给出用 while 语句的解。

例 4.10

```
/* 用 while 语句将输入的整数位颠倒输出 */
main()
{
    int i, r ;
    printf("Input an integer\n") ;
    scanf("%d", &i) ;
    if (i == 0)
        printf("%d", i) ;
    while (i ! = 0) {
        r = i % 10 ;
```

```
        printf("%d", r);
        i /= 10;
    }
    printf("\n");
}
```

此程序与例 4.9 完成同样的功能。

例 4.10 中的 if 语句是处理当 i 为 0 的情况。如果没有此句，当输入为 0 值(一个合法的输入值)时，没有数值输出。比较一下例 4.9 和例 4.10，显然例 4.9 更方便，也更自然。

当 do—while 循环的循环体中只有一条语句时，不需要用花括号括起，如可以把

```
do {
    r = i % 10;
    printf("%d", r);
} while ((i /= 10) != 0);
```

写成

```
do
    printf("%d", i % 10);
while ((i /= 10) != 0);
```

但这样写表达得不很清楚，可能会使性急的程序员一看到 while 就认为是一个 while 循环的开始，为了避免这种情况，也为了程序更易理解，多数程序员总倾向于用花括号括起 do—while 语句的循环体，即使只有一条语句。于是上面的语句可以写成

```
do {
    printf("%d", i % 10);
} while ((i /= 10) != 0);
```

比较一下，是不是清楚一些?

第四节　循环的嵌套

一个循环语句的循环体中又可以包含循环语句，这种循环中有循环的结构就是循环的嵌套。

例 4.11 求 100 以内的全部素数。

我们可以用最简单的办法来求素数，即一个数 m 如果不能被 2～m－1 之间的任何数整除的话，就表明它是一个素数，程序如下：

```
/*求 100 以内的素数*/
main()
{
    int m, n, i, prime;
    i = 0;
    for (m = 2; m <= 100; m ++) {
        prime = 1; /*设一个标志，先假定是素数*/
```

```
        for (n = 2; n < m; n ++)
            if (m % n == 0)
                prime = 0 ; /* 表明 m 不是素数 */
        if (prime) { /* 假定值没有改变 */
            printf("%6d", m) ;
            i ++ ;
            if (i % 5 == 0) /* 每行输出 5 个值 */
                printf("\n") ;
        }
    }
    if (i %5 ! = 0)
        printf("\n") ;
}
```

运行结果

```
2     3     5     7     11
13    17    19    23    29
31    37    41    43    47
53    59    61    67    71
73    79    83    89    97
```

这个程序有两重循环，在内层循环中，对每个给定的 m 值用 2～m－1 去除，判别是否能够整除。而外层循环则一个个地提供 m 值，从 2 到 100。在程序中，我们设置了一个标志 prime，对于任何一个被测试的数来说，先假定它是素数，置 prime 为真(1)，如果它能被 2～m－1 中某个数整除，说明它不是素数，置标志 prime 为假(0)，如果内层循环没有置 prime 为 0，则表明 m 不能被 2～m－1 中任何数整除，说明 m 是素数，这时的 prime 仍然是 1。要注意，对于参加测试的每一个数都是如此，因此 prime 的初值应设置在外层循环以内，内层循环以外。

例 4.11 给出了 for 循环嵌套的例子。实际上，三种循环都是可以互相嵌套的，下面是一些合法的循环嵌套形式

```
(1)while(){
⋮
while(){
⋮
}
⋮
}
(2)do{
⋮
```

```
while(){
 ⋮
}
 ⋮
}while()
(3)while(){
 ⋮
for(){
 ⋮
}
 ⋮
}
(4)for(){
 ⋮
do{
 ⋮
}while()
 ⋮
}
(5)while(){
 ⋮
do{
 ⋮
}while()
 ⋮
for(){
 ⋮
}
 ⋮
}
(6)for(){
 ⋮
while(){
 ⋮
for(){
 ⋮
}
 ⋮
}
```

```
    ⋮
}
```

循环允许嵌套，if 语句允许嵌套，循环与条件语句之间也可以嵌套，程序就这样由简单的语句构造成各种结构。由于任何复杂的程序都可以用顺序、判定、循环三种结构实现，所以，你应该熟练地掌握循环控制结构和判定控制结构，并能够灵活地用它们来解决实际问题。

程序结构复杂起来，经常需要用到花括号，许多初学者往往写了左侧的花括号，而忘记右侧的花括号，以致造成错误。缩进书写是减少花括号丢失的一个办法，另一个不易出错的办法是：上机时总是同时给出左右两个花括号，然后再在其中添加所需的内容。

第五节　break 语句

break 语句的形式很简单：

break ;

在前一章介绍 switch 语句时，我们已经看到了 break 语句，它的功能是终止其后语句的执行，退出 switch 语句。break 语句的另一个用途是使一个循环立即结束，也就是说在循环中遇到 break 语句时，循环立即终止，程序转到循环体后的第一个语句去继续执行。

例 4.12

```
main()
{
    int i ;
    for (i = 1; i <= 100; i ++) {
        printf("%d ", i) ;
    if (i == 10)
        break ;
    }
    printf("\n") ;
}
```

运行结果为

```
1  2  3  4  5  6  7  8  9  10
```

虽然 for 语句规定的循环是 i 从 1 到 100，但当 i 等于 10 时，执行一个 break 语句，此时 break 强制从循环中跳出而不继续去判断 i 是否小于等于 100。

下面我们再来看一个用 break 语句退出恒真条件循环的例子。

例 4.13　求调和级数中第多少项的值大于 10。

所谓调和级数的第 n 项形式为

$$1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5}+\cdots+\frac{1}{n}$$

我们要求的就是使值大于 10 的最小的 n，程序如下：

```
#define LIMIT 10
main()
{
    int n ;
    float sum ;
    sum = 0.0 ;
    n = 1 ;
    for ( ; ; ) {
        sum = sum + 1.0 / n ;
        if (sum> LIMIT)
            break ;
        n ++ ;
    }
    printf("n=%d\n", n) ;
}
```

运行结果

n=12367

这里 for 循环不判断终止条件，如果循环体中没有退出循环的语句，循环将无休止地进行下去，而 break 语句的设置正是为了在满足一定条件后，程序能从循环中退出。

从上面的例子可以看出，break 语句在循环体中使用时，总是与 if 一起使用，当条件满足(或不满足)时，负责退出循环。要注意，如果循环体中使用 switch 语句，而 break 出现在 switch 语句中，则它只用于结束 switch，而不影响循环。

break 语句只能结束包含它的最内层循环，而不能跳出多重循环，如

```
for(){
 ⋮
while(){
 ⋮
if()
break;
 ⋮
}
 ⋮
}
```

break 语句的执行使程序从内层 while 循环中退出，继续执行 for 循环的其它语句，而不是退出外层循环。

第六节　continue 语句

continue 语句的形式也很简单：

continue ;

它只能出现在循环体中，其功能是立即结束本次循环，即遇到 continue 语句时，不执行循环体中 continue 后的语句，立即转去判断循环条件是否成立。

continue 与 break 语句的区别在于：continue 只是结束本次循环，而不是终止整个循环语句的执行，break 则是终止整个循环语句的执行，转到循环语句后下一条语句去执行。我们可以看看下面两个循环结构：

```
(1)while(表达式 1){          (2)while(表达式 1){
        ⋮                            ⋮
    if(表达式 2)                 if(表达式 2)
      break;                       continue;
        ⋮                            ⋮
    }                            }
```

它们的流程如图 4-4 和图 4-5 所示。

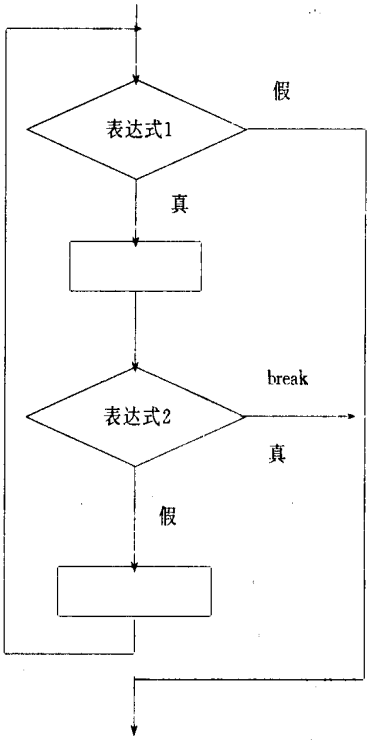


图4-4 break结构

例 4.14 求输入的正数之和。

```
main()
{
    int i, n, sum ;
    for (i = 1; i <= 10; i ++) {
```

```
        scanf("%d", &n);
        if (n < 0)
            continue;
        sum = sum + n;
    }
    printf("sum=%d\n", sum);
}
```

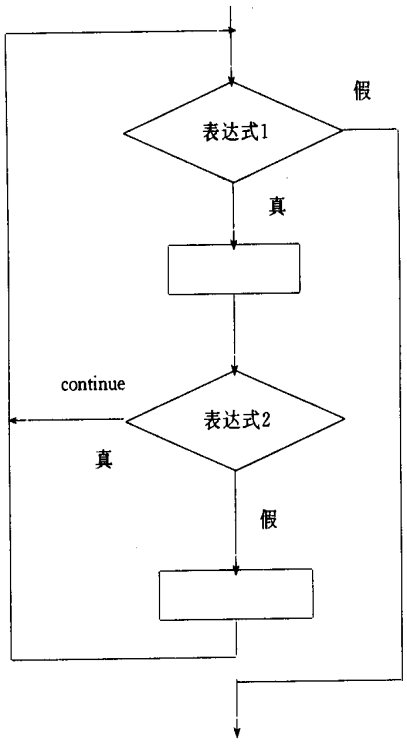


图4-5 continue结构

第七节 goto语句和标号

goto语句是无条件转向语句，其一般形式为：

goto 语句标号

语句标号用标识符表示，用来表示程序的某个位置。goto语句的功能是无条件地把程序转到标号所在的位置。

计算机语言中的goto语句曾引起过很大争议，许多人主张不使用goto语句，理由是它使程序难于控制，也大大降低了可读性。而另一些人主张保留goto语句，因为它在解决一些特定问题时很方便。这几年人们倾向有保留地使用goto语句，前提是如果可以使程序更清晰的话。实际上这是对熟练的程序员而言的，对于初学者来说，最好是不使用goto语句。曾有人对goto语句做过形象的比喻，它是一种双面刀，当攻击敌人时，也可能会伤害到自己，所以只有高手才可以使用。本书的原则是，除此节之外将不再出现goto语句，而且甚至在本节也不提供完整的例子。

goto 语句往往用来从多重循环中跳出，如

```
while(){
    ⋮
    while(){
        ⋮
        for(){
            ⋮
            if()
                goto stop ;
            ⋮
        }
        ⋮
    }
    ⋮
}
stop :
printf("error\n") ;
```

这里谨慎地使用了 goto 语句，用以跳出多重循环，如果不用它而用 break，则要做多次测试，如

```
flag = 0 ;
while () {
    ⋮
    while () {
        ⋮
        for () {
            ⋮
            if () {
                flag = 1 ;
                break ;
            }
            ⋮
        }
        if (flag)
            break ;
        ⋮
    }
    if (flag)
        break ;
    ⋮
```

```
}
```

这样写程序，可读性反而差了。

第八节　常见错误

1. 误把＝作为等号使用。

这与条件语句中的情况一样，如

```
while (x = 1) {
⋮
}
```

这是一个恒真条件的循环，正确地写法应是

```
while (x == 1) {
    ⋮
}
```

2. 忘记用花括号括起循环体中的多个语句，这也与条件语句类似，如

```
while (i <= 10)
    printf("%d", i) ;
    i ++ ;
```

由于没有用花括号，循环体就只剩下 printf("%d", i) 一条语句。正确地写法应为

```
while (i <= 10) {
    printf("%d", i) ;
    i ++ ;
}
```

3. 在不该加分号的地方加了分号。如

```
for (i = 1; i <= 10 ;i ++) ;
    sum = sum + i ;
```

由于 for 后加了一个分号，表示循环体只有一个空语句，而 sum = sum + i ;与循环无关。正确的写法应是

```
for (i = 1 ;i <= 10; i ++)
    sum = sum + i ;
```

4. 花括号不匹配。

由于各种控制结构的嵌套，有些左右花括号相距可能较远，这就可能会忘掉右侧的花括号而造成花括号不匹配，这种情况在编译时可能产生许多莫名其妙的错误，而且错误提示与实际错误无关。解决的办法可以是在括号后加上表示层次的注释，如

```
while () { /*(1)*/
    ⋮
    while () { /*(2)*/
        ⋮
        if () { /*(3)*/
```

```
            ⋮
            for () { /*(4)*/
                ⋮
            } /*(4)*/
        } /*(3)*/
        ⋮
        for () { /*(3)*/
            ⋮
        } /*(3)
        ⋮
    } /*(2)*/
    ⋮
} /*(1)*/
```

每次遇到嵌套左括号时就把层次加1,每次遇到右括号时就把层次减1,当括号不匹配时最后的右括号的层次号就不是1,可以肯定有括号丢失。

5. 死循环。

由于某种原因使循环无休止的运行,或直到出错才结束循环,如

```
i = 1 ;
while (i <= 10)
sum = sum + i ;
```

由于i没有改变,所以i<=10永远为真,循环将一直延续下去。另一种情况是,虽然有改变循环条件的运算,但改变的方向不对,如

```
i = 1 ;
while (i >= 0) {
    sum = sum + i ;
    i ++ ;
}
```

i开始就大于0,而以后每次都增加i的值,使条件i>=0总是成立,直到i值为32767后再加1,超越正数的表示范围而得到负值时才结束,这时的结果肯定与希望的不同。

再有一种情况是循环条件被跳过去造成的,如

```
for (i = 1; i == 10; i += 2) {
    ⋮
}
```

由于i值每次增加2,所以取值为1,3,5,7,9,11…把10跳过去了,正确的写法应为

```
for (i = 1; i <= 10; i += 2) {
    ⋮
}
```

当i值超过10时循环就结束了。

第五章　数　　组

本章学习重点

本章介绍数组的概念和使用，并专门讨论可以看作是字符串的一维字符数组。学习本章应掌握如何利用数组来求解应用问题。

我们前面介绍过的整型、浮点型、字符型等都是基本数据类型，除此之外，C 语言中还有由基本类型数据按一定规则组织起来的构造类型数据。这些构造数据类型有数组、结构、联合和枚举。本章中我们介绍数组类型。

数组是同类型数据的有序集合，一个数组中的所有数据都来自同一类型，如一个正文行可以看成是一个由字符组成的字符数组，一个向量可以看成是由整数或浮点数组成的数组，一个矩阵可以看作是由向量组成的数组。数组用一个统一的名字来标识，每一个分量可用数组名和下标来唯一的确定。

第一节　一维数组

5.1.1　一维数组的定义与使用

在实际中，我们经常要处理一组有关系的数据，比如说一组学生的成绩，我们可以定义它们为整数，如

```
int grade0, grade1, grade2, grade3, grade4 ;
```

当学生的成绩较多时，比如说 100 个，这样定义很不方便，而且各变量之间也没有制约关系，不能保证它们是一组相关联的变量，使用数组我们可以这样写：

```
int grade[100] ;
```

这就定义了一个整型数组，它有 100 个分量，我们可用它来存放 100 个学生成绩，这样表示既清楚又方便。

一维数组的定义形式为

类型标识符　数组名[常量表达式]

例如：

```
int a[10] ;
float f[50] ;
char name[20] ;
```

都是正确形式的数组定义。

数组元素在内存中连续存放，如

```
int a[10] ;
```

定义一个由 10 个元素组成的整型数组 a，它在内存中存放的形式如图 5.1 所示。

图 5-1 数组的存放形式

其中每一个元素都相当一个整型变量，可以存放一个整型数值。与 10 个整型变量不同之处在于：数组元素是按顺序排列的，这在处理数据时会带来许多方便。

从定义中可以看出，数组名是一个标识符，其后方括号中的常量表示数组元素的个数，C 语言中数组元素的下标总是从 0 开始，因此，数组 a 的元素如图 5.1，是从 a[0]到 a[9]，同样理由，定义

```
float f[50];
```

规定数组 f 的元素是从 f[0]到 f[49]。

像基本类型的变量一样，数组也必须先定义后使用。

例 5.1

```
main()
{
    int a[5], sum;
    a[0] = 3;
    a[1] = 1;
    a[2] = 7;
    a[3] = 4;
    a[4] = 8;
    sum = a[0] + a[1] + a[2] + a[3] + a[4];
    printf("sum=%d\n", sum);
}
```

运行结果

```
sum=23
```

这里我们看到了数组元素的使用，一个数组元素是由数组名和下标值来标识的，如 a[0]、a[1]等。数组元素的一般引用形式为

数组名[下标表达式]

虽然数组定义和数组元素的引用形式类似，但它们含义不同，定义

```
int a[5];
```

表示数组 a 共有 5 个元素，而

```
a[5]
```

表示 a 数组中下标为 5 的元素。

在例 5.1 中数组元素象普通变量一样使用，而实际中，由于数组元素排列的规律性，我们可以通过其下标值，用循环的办法来操作数组。

例 5.2　求一组学生的平均成绩

```
#define NUM 5
main()
{
    int grade[NUM], i, total ;
    float average ;
    total = 0 ;
    printf("please input %d scores\n", NUM) ;
    for (i = 0; i < NUM; i ++){
        printf("grade[%d]=", i) ;
        scanf("%d", &grade[i]) ;
        total = total + grade[i] ;
    }
    average = (float) total / NUM ;
    printf("average=%0.2f\n",average) ;
}
```

运行结果

```
please input 5 scores
grade[0]=90
grade[1]=65
grade[2]=97
grade[3]=57
grade[4]=78
average=77.40
```

这个程序从键盘上接收 5 个成绩，求平均成绩，for 循环控制变量 i 从 0 变化到 4，printf 语句每次显示出要接收的下标变量 grade[0]～grade[4]，scanf 接收相应的值，再把它们加到 total 中。

要注意，这里 scanf 不能一次接收整个数组的值，如写成

```
scanf("%d", &grade) ;
```

是错误的，要想对数组中所有的元素赋值，就要通过循环。

引入数组后，许多需要循环处理的问题就变得更方便解决了，下面我们再来看一下求菲波那契数列的问题，这次用数组。

例 5.3

```
/* 用数组求解 Fabonacci 数列 */
#define NUM 20
#define COL 5
main()
{
    int f[NUM + 1], i ;
```

```
    f[1] = 0 ;
    f[2] = 1 ;
    for (i = 3 ; i <= NUM ; i ++)
        f[i] = f[i - 2] + f[i - 1];
    for (i = 1; i <= NUM; i++){
        printf("%10d", f[i]) ;
        if(i % COL == 0)
            printf("\n") ;
    }
}
```

运行结果为

```
    0       1       1       2       3
    5       8      13      21      34
   55      89     144     233     377
  610     987    1597    2584    4181
```

与例 4.7 完全一样，比较一下两个例子，这个程序用 f_{n-2}和 f_{n-1}求 f_n，写成

```
f[n] = f[n-2] + f[n-1] ;
```

就显得非常自然，容易书写，也更容易阅读理解。

程序中定义数组元素个数为 NUM+1(21 个)，使用时没有用 f[0]，而只使用了 f[1]~f[20]，这样和数学公式表达相一致，处理也更方便(否则每行输出 5 个值就要麻烦一些)。

有了数组，许多问题的处理变得容易起来，但 C 语言的数组也有一个令人很头疼的地方，就是它不作下标是否越界的判断，如例 5.3 中第二个 for 语句写成

```
for (i = 1; i < NUM + 1 ; i ++)
```

循环体中的 f[i]在最后一次将是 f[21]，而我们的定义

```
int f[NUM + 1] ;
```

相当于

```
int f[21] ;
```

f 的下标只能是 0~20，f[21]应该是个错误，但很遗憾，C 语言不管那么多，它继续执行，于是多计算了一个值，而这个值本来是不应该产生的，并且有时还会产生合理的结果，这真使人哭笑不得。由于 C 语言的设计者对下标越界这么宽容，所以就只好我们自己费心了。

虽然有上面的缺点，但数组的使用还是给我们解决实际问题带来许多方便，有些问题不用数组甚至会感到无从下手。

5.1.2 一维数组应用举例

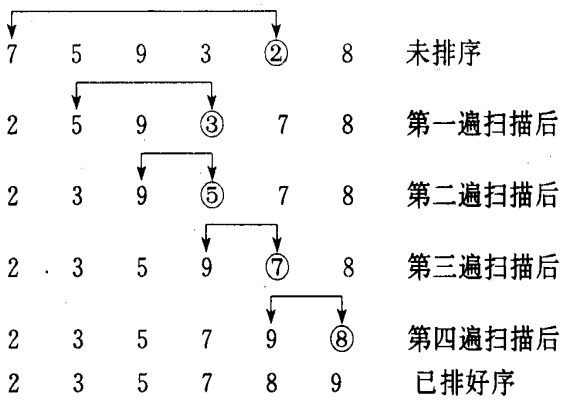
例 5.4 选择排序

所谓排序是指把一组杂乱无章的数按照大小顺序排列，选择排序采用的办法是：首先找出值最小的元素，然后把这个元素与第一个元素互换，这样值最小的元素就放到了第一个位置，接着，再从剩下的元素中找值最小的，把它和第二个元素互换，使得第二小的元素放在第

二个位置上，依此类推，直到所有的值由小到大顺序排列为止。如有数据

5　9　3　2　8

则选择排序的过程如下



程序如下：

```
#define NUM 10
main()
{
    int a[NUM], i, j, r, temp ;
    printf("Please input %d numbers\n", NUM) ;
    for (i = 0; i < NUM ; i ++)
        scanf("%d", &a[i]) ;
    for (i = 0; i < NUM - 1; i ++) { /* 扫描 NUM-1 次 */
        r = i ;
        for (i = i + 1; j < NUM ;j ++) /* 在一遍扫描中 */
            if (a[i] < a[r] ) /* 找最小的元素 */
                r = j ;
        if (r ! = i) { /* 互换 */
            temp = a[i] ;
            a[i] = a[r] ;
            a[r] = temp ;
        }
    }
    printf("The array after sort:\n") ;
    for (i = 0; i < NUM; i++)
        printf("%5d", a[i]) ;
    printf("\n") ;
}
```

运行结果

```
Please input 10 numbers
9  53  -3  100  -21  0  72  44  -99  60
```

The array after sort:

−99 −21 −3 0 9 44 53 60 72 100

例 5.5 用筛法求素数

用筛法求素数的基本思想是:把某一范围内的正整数按从小到大顺序排列,宣布 1 不是素数,把它筛掉。然后从剩下的数中选择最小的,宣布它是素数,并去掉它的倍数,而后再从剩余的数中选最小的,宣布为素数,并去掉这个数的倍数,依次类推,直到筛子为空时结束。如有

2 3 4 5 6 7 8 9 10

11 12 13 14 15 16 17 18 19 20

21 22 23 24 25 26 27 28 29 30

首先宣布 2 是素数,然后去掉它的倍数

② 3 ④ 5 ⑥ 7 ⑧ 9 ⑩ 11

⑫ 13 ⑭ 15 ⑯ 17 ⑱ 19 ⑳ 21

㉒ 23 ㉔ 25 ㉖ 27 ㉘ 29 ㉚

余下的数是

3 5 7 9 11 13 15 17 19 21 23 25 27 29

宣布最小的数 3 是素数,并去掉它的倍数。

③ 5 7 ⑨ 11 13 ⑮ 17 19 ㉑ 23 25 ㉗ 29

如此下去直到所有的数都被筛完,宣布的素数有

2 3 5 7 11 13 17 19 23 29

写程序时,采用一个数组,用数组的下标表示自然数,如果一个数不在筛中就将其对应的元素值赋 0,如果仍在筛中,则那个元素值为 1。程序如下:

```
#define RANGE 200
main()
{
    char sieve[RANGE + 1];
    int i, j, count;
    for (i = 0; i <= RANGE; i ++)
        sieve[i] = 1; /* 初始化 */
    sieve[0] = sieve[1] = 0; /* 0 和 1 不是素数 */
    count = 0;
    for (i = 2; i <= RANGE; i ++)
        if (sieve[i] == 1) { /* i 是素数 */
            printf("%5d", i); /* 输出素数 */
            count ++;
            if (count % 8 == 0) /* 每行输出 8 个值 */
                printf("\n");
            for (j = i; j < RANGE; j += i)
```

```
                sieve[j] = 0 ; /* 筛去 i 的倍数 */
        }
    printf("\n") ;
}
```

运行结果

```
  2    3    5    7   11   13   17   19
 23   29   31   37   41   43   47   53
 59   61   67   71   73   79   83   89
 97  101  103  107  109  113  127  131
137  139  149  151  157  163  167  173
179  181  191  193  197  199
```

例 5.6 求杨辉三角形。

所谓杨辉三角形就是二次项的系数

```
1
1  1
1  2  1
1  3  3  1
1  4  6  4  1
⋮
```

我们编一个程序来求解它们，并以上面的形式输出。杨辉三角形的头一行是容易生成的，只需简单地赋值就可以了。那么对于任意一行呢？我们可以看到，第一列和最后一列总是 1，而其它数是上面一行中本列和前一列元素之和，如

```
1  4  6  4  1
```

一行

```
4=3+1
6=3+3
4=1+3
```

这样我们可以利用两个数组，用存放在前一个数组中的数据生成新的一行放在另一个数组中，通过来回交换就可生成一行行的数据。我们也可以只用一个数组，每次倒着生成数组中的各元素，如

```
yanghui[4]=yanghui[4]+yanghui[3];
yanghui[3]=yanghui[3]+yanghui[2];
yanghui[2]=yanghui[2]+yanghui[1];
```

本程序采用后一种方法，程序如下：

```
/*求杨辉三角形*/
#define LASTROW 10
main()
```

```
{
    int yanghui[LASTROW + 1], row, col ;
    yanghui[0] = 1 ;
    printf("%d\n", yanghui[0]) ; /* 输出头一行 */
    /* 由前一行生成新的一行 */
    for (row = 1; row <= LASTROW; row ++) {
        yanghui[row] = 1 ;
        for (col = row - 1; col > 0; col --)
            yanghui[col] = yanghui[col] + yanghui[col - 1] ;
        for (col = 0; col <= row; col ++)
            printf("%4d",yanghui[col]) ; /* 输出一行 */
        printf("\n") ;
    }
}
```

运行结果

```
   1
   1   1
   1   2   1
   1   3   3   1
   1   4   6   4   1
   1   5  10  10   5   1
   1   6  15  20  15   6   1
   1   7  21  35  35  21   7   1
   1   8  28  56  70  56  28   8   1
   1   9  36  84 126 126  84  36   9   1
   1  10  45 120 210 252 210 120  45  10   1
```

5.1.3 一维数组的初始化

前面的例子中，都是通过赋值或输入的方式给数组元素所需要的值，我们也可以在定义时给数组元素赋初始值，如

```
static int a[10] = {9,8,7,6,5,4,3,2,1,0} ;
```

这里 static 是静态存储的意思，我们将在第六章中介绍它的概念。C 语言规定：只有静态数组和外部数组才能够初始化。

数组元素的初始化值按照顺序放在花括号中，经过上面的定义和初始化后，a[0]=9，a[1]=8，a[2]=7，…a[9]=0，在内存中它们的形式如图 5-2 所示。

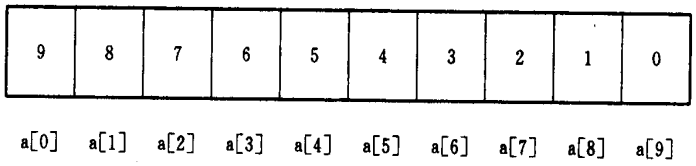


图 5-2 数组的初始化

初始化时也可只对部分元素赋值，如

static int a[10] = {9,8,7,6,5} ;

把值 9、8、7、6、5 赋给 a 的前 5 个元素，后 5 个元素系统自动赋值给 0 值，情况如图 5.3 所示。

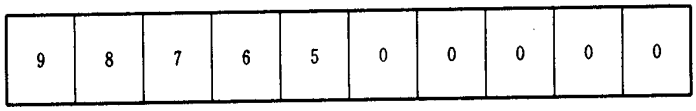


图 5-3 数组的存储形式

实际上，对于静态数组和外部数组，如不显式地初始化，系统将自动对所有元素赋初值 0。

如果对数组的全部元素都初始化，则可以不指定数组长度，如

static int a[10] = {9,8,7,6,5,4,3,2,1,0} ;

也可以写成

static int a[] = {9,8,7,6,5,4,3,2,1,0} ;

这时由初值的个数决定数组的长度。

第二节 多维数组

5.2.1 多维数组的定义

具有多个下标的数组称为多维数组，其中最简单的是二维数组，其一般形式为

类型标识符 数组名[常量表达式][常量表达式]

例如

int m[5][5] ;

定义了一个具有 5 行 5 列的整型数组 m。

C 语言把二维数组看成是一个数组的数组。这样 m 的元素 m[0]，m[1]～m[4]实际上都是一个一维数组，它们各自有 5 个元素，如图 5-4 的所示。

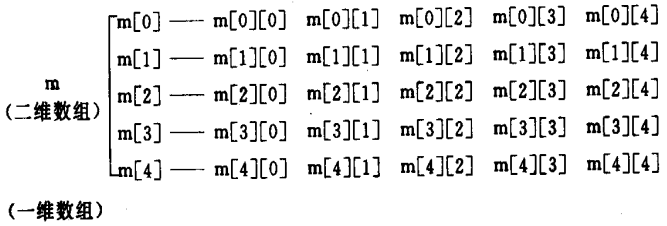


图 5-4 二维数组的结构

由这种形式我们可以了解更高维的数组。一个三维数组实际上可以看做是元素为二维数组的数组，依此类推。

在 C 语言中，二维数组的元素是按行存储的，即在内存中是先放第一行的元素，再放第二行的元素…。图 5.5 给出了二维数组 m 的排列顺序。

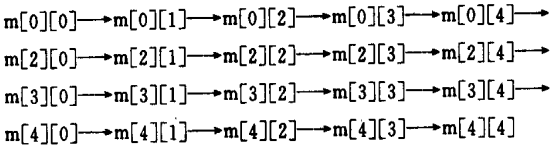


图 5-5 二维数组元素的排列顺序

5.2.2 多维数组的初始化

多维数组同样可以初始化，如对二维数组 m 可以写

```
static int m[5][5]={0,0,0,0,0,1,1,1,1,1,2,2,2,2,2,3,3,3,3,3,4,4,4,4,4};
```

为了表达清楚，我们通常把每一行用一个花括号括起，并且在书写时单独占一行，如

```
sttic int m[5][5] = { {0,0,0,0,0},
                      {1,1,1,1,1},
                      {2,2,2,2,2},
                      {3,3,3,3,3},
                      {4,4,4,4,4}} ;
```

显然这样表示要比前一种表示清楚得多。

如果初始化值比数组元素数少，则后面的不足者认为值是 0，如

```
static int a[4][5] = {{1,2},
                      {},
                      {0,3,1},
                      } ;
```

初始化后数组的各元素值为

```
1  2  0  0  0
0  0  0  0  0
0  3  1  0  0
0  0  0  0  0
```

对多维数组初始化时，如果提供全部初始化值，则第一维的下标可以省略，但后面各维下标不可缺少，如

```
static int a[3][4] = { {1, 2, 3, 4},
                       {5, 6, 7, 8},
                       {9,10,11,12}} ;
```

等价于

```
static int a[][4] = {1,2,3,4,5,6,7,8,9,10,11,12} ;
```

如果初始化值中有较多的 0，用下面的方式可以省略第一维。

```
static int a[][4]={ {0,2},
                    {  },
                    {0,0,1,1}} ;
```

5.2.3 多维数组应用举例

例 5.7 矩阵转置

矩阵转置是把矩阵的行和列互换，如一个 3×4 的矩阵

```
1    2    3    4
5    6    7    8
9    10   11   12
```

转置后变成 4×3 的矩阵

```
1    5    9
```

2 6 10

3 7 11

4 8 12

程序如下：

```
#define ROW 3
#define COL 4
main()
{
    int a[ROW][COL], b[COL][ROW] ;
    int i ,j ;
    printf("Please input element of the matrix  a") ;
    printf("(%d×%d)\n", ROW, COL) ;
    for (i = 0; i < ROW; i ++) /*输入矩阵 a*/
        for (j = 0; j < COL; j ++)
            scanf("%d", &a[i][j]) ;
    for (i = 0; i < ROW; i ++) /*转置*/
        for (j = 0; j < COL; j ++)
            b[j][i] = a[i][j] ;
    printf("Matrix b:") ;
    for (i = 0; i < COL; i ++) { /*输出转置后的矩阵*/
        for (j = 0; j < ROW; j ++)
            printf("%5d", b[i][j]) ;
        printf("\n") ;
    }
}
```

运行结果

Please input elements of the matrix a(3×4)

1 2 3 4

5 6 7 8

9 10 11 12

Matrix b:

1 5 9

2 6 10

3 7 11

4 8 12

例 5.8 求一个 n×n 矩阵的转置。

在上一个例子中，由于矩阵的行列不同，必须使用两个数组，对于 n×n 矩阵，我们可以只用一个数组，在数组中进行行列元素的互换。程序如下：

```
#define ROW 4
main()
{
    int sm[ROW], i, j, temp ;
    printf("Input elements of a matrix(%d×%d)\n",ROW,ROW);
    for (i = 0; i < ROW; i ++)
        for (j = 0 ;j < ROW; j++)
            scanf("%d",&sm[i][j]) ;
    for (i = 0; i < ROW - 1; i++) /*转置*/。
        for (j = i + 1; j < ROW; j ++) {
            temp = sm[i][j] ;
            s[i][j] = s[j][i] ;
            s[j][i] = temp ;
    }
    printf("The matrix has been transposed:\n") ;
    for (i = 0; i < ROW; i ++) {
        for (j = 0; j < ROW; j ++)
            printf("%5d",sm[i][j]) ;
        printf("\n") ;
    }
}
```

运行结果

```
Input elements of a matrix(4×4)
    1    2    3    4
    5    6    7    8
    9    10   11   12
    13   14   15   16
The mratrix has been transposed
1     5     9     13
2     6     10    14
3     7     11    15
4     8     12    16
```

这个程序中，利用了矩阵的上三角与下三角对换，循环控制

```
for (i = 0; i < ROW - 1; i ++)
for (j = i + 1; j < ROW; j ++)
```

使得 j 总是大于 i，即列的值大于行的值，这是上三角，然后是

```
temp = sm[i][j] ;
sm[i][j] = sm[j][i]
sm[j] ;[i] = temp ;
```

也就是 sm[i][j]和 sm[j][i]互换：

```
sm[i][j]⇔sm[j][i]
```

这样上三角的内容就被换到下三角来，完成了矩阵的转置。如果将循环写成

```
for (i = 0; i < ROW; i ++)
for (j = 0; j < ROW; j ++)
```

则上三角的元素与下三角的元素交换后，下三角的元素又与上三角对应的元素交换，两次对换，等于没有转置。

例 5.9　求解幻方问题。

幻方是一种古老的数学游戏，n 阶幻方就是把整数 1～n^2 排成 n×n 的方阵，使得每行中的各元素之和，每列中的各元素之和，以及两条对角线上的元素之和都是同一个数 S，S 称为幻方的幻和。在中世纪的欧洲，对幻方有某种神秘的观念，许多人佩带幻方以图避邪。奇数阶幻方的构造方法很简单，我们先来看一个三阶幻方，如图 5.6 所示。

8	1	6
3	5	7
4	9	2

图 5-6　三阶幻方

各数在方阵中的位置可以这样确定：

首先把 1 放在最上一行正中间的方格中，然后把下一个整数放置到右上方，如果到达最上一行，下一个整数放在最后一行，就好象它在第一行的上面，如果到达最右端，则下一整数放在最左端，就好象它在最右一列的右侧，当到达的方格中已填上数值时，下一个整数就放在刚填写上数码的方格的正下方。照着三阶幻方，从 1 至 9 走一下，就可以明白它的构造方法，下面是程序。

```
/*求奇数阶幻方   */
#define MAX 15
main()
{
    int m, mm, i, j, k, ni, nj ;
    int magic [MAX][MAX] ;
    printf("Enter the number you wanted\ n");
    scanf("%d", &n) ;
    for (i = 0;i < m; i ++) /*初始化*/
        for (j = 0; j < m; j ++)
            magic[i][j] = 0 ;
    if ( (m > 0) && (m % 2 ! = 0) ) { /*奇数阶*/
        mm = m * m ;
```

```
        i = 0 ; /* 第一个值的位置 */
        j = m / 2 ;
        for (k = 1; k <= mm; k ++) {
            magic[i][j] = k ;
            /* 求右上方方格的坐标 */
            if (i == 0) /* 最上一行 */
                ni = m - 1 ; /* 下一个位置在最下一行 */
            else
                ni = i - 1 ;
            if (j == m - 1) /* 最右端 */
                nj = 0 ; /* 下一个位置在最左端 */
            else
                nj = j + 1 ;
            /* 判断右上方方格是否已有数 */
            if (magic[ni][nj] == 0) { /* 右上方无值 */
                i = ni ;
                j = nj ;
            }
            else /* 右上方方格已填上数 */
                i ++ ;
        }
        for (i = 0; i < m; i ++) {
            for (j = 0; j < m; j ++)
                printf("%4d", magic[i][j]) ;
            printf("\n") ;
        }
    }
    else /* m<=0 或 m 是偶数 */
        printf("Error in input data. \n") ;
}
```

运行结果

```
Enter the number you wanted
5
17   24    1    8   15
23    5    7   14   16
 4    6   13   20   22
10   12   19   21    3
```

11　18　25　2　9

第三节　字符数组

5.3.1　字符数组的定义及初级化

在C语言中，一维的字符型数组有着特殊的地位，我们把它简称为字符数组。字符数组中的每一个元素可存放一个字符，我们把字符数组中的字符看成一个整体，就相当于一个字符串。通常C语言处理字符串就是利用字符数组的形式。

字符数组的定义没有什么太特殊的地方，如

char s[10]；

定义了一个有10个元素的字符型数组，一般我们说成：长度为10的字符数组。

字符数组也可以初始化，如

static char str[16] = {'T','h','i','s',' ','i','s',' ','a',' ','s','t','r','i','n','g'}；

对数组的各元素分别赋予字符值，赋初值后的情形如图5-7所示。

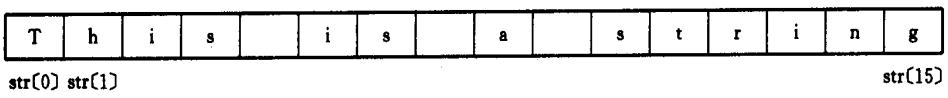


图5-7　字符数组赋初值

如果花括号中提供的初值个数超过数组长度，将产生错误，如果初值个数少于数组长度时，剩余的元素自动用空格补足，如

static char c[10] = {'c',' ','p','r','o','g','r','a','m'}；

图5.8给出了数组C赋值初值后的状态。

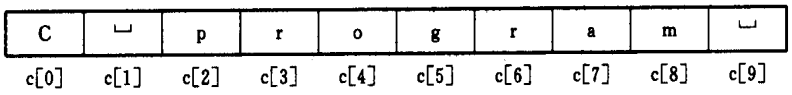


图5-8　字符数组部分赋初值

字符数组的长度也可用初值来确定，如

static char str[] = {'T','h','i','s',' ','i','s',' ','a',' ','s','t','r','i','n','g'}；

将字符数组str的长度定义为16。

用一个一个的字符常量来初始化字符数组实在是麻烦，能不能用一个字符串常量来初始化呢？C语言的设计者已经想到了这一问题，于是我们可以定义

static char str[] = {"This is a string"}；

这时我们定义了一个长度为17的数组。为什么是17呢？别忘了字符串常量的最后总带有一个字符结束标志\0，这时的内存情况如图5-9所示。

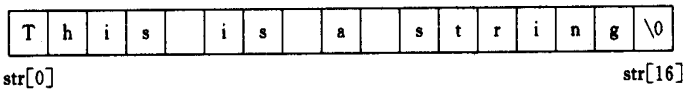


图5-9　用字符串常量初始化字符数组

初始化的形式还可以进一步简化，写成

static char str[]="This is a string" ;

这种形式使用得更多。

当然你也可以用字符串常量来初始化一个已明确指出长度的字符数组，如

static char str[10] = "string";

数组 str 在内存中的形式如图 5-10 所示。

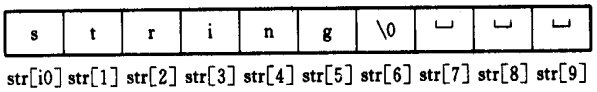


图 5-10 用字符串常量初始化确定长度的字符数组

5.3.2 字符数组应用举例

例 5.10 求字符串的长度。

```
#define MAXLEN 80
main()
{
    char str[MAXLEN + 1] ;
    int l ;
    printf("Input a string\n");
    scanf("%s",str) ;
    l = 0 ;
    while (str[l] ! = '\0')
        l ++ ;
    printf("The length of this string is %d\n",l);
}
```

运行结果

```
Input a string
hello
The length of this string is 5
```

在本程序的 scanf 输入使用了转换字符 s，s 用来输入一个字符串，输入项这时用数组名，并且前面不带 &。如果写成

scanf("%s",&str) ;

反而错了。C 编译对数组名的处理是：把它作为存放数组的内存的起址地址。由于数组名本身就代表地址，因此也就不需要用求地址运算符 & 了。

scanf 接收一个字符串并自动为它加上一个结束标志\0，因此我们可以利用\0 来判断字符串的长度。while 循环中，当字符不是结束标志时，下标值就逐渐增加，相当于在字符串中逐个后移，直到遇到\0，此时的下标值也就是字符串的长度。

例 5.11 从键盘上读入一个输入行并输出。

scanf 中使用 s 转换字符可以读入一个字符串，当遇到空白符时就认为字符串结束了，因此不能用它来读入一个输入行，因为一行中可能有空格。我们的程序要逐个检查输入的字符，只有遇到换行符\n 时才停止读入，程序如下：

```
/*读入一个输入行*/
#include "stdio.h"
#define MAXLEN 80
main()
{
    char line[MAXLEN + 1], c ;
    int i ;
    i = 0 ;
    while ( (c = getchar() ) != '\n')
        line[i ++] = c ;
    line[i] = '\0' ;
    printf("%s\n", line) ;
}
```

运行结果

```
How do you do
How do you do
```

这个程序用 getchar 逐个输入字符，当不是换行符时，就把读入的字符送到字符数组 line 中，i++使字符数组每得到一个字符后，就将下标后移一格，准备存放下一个字符。当遇到\n 时，不把\n 送进数组 line，而是在最后加一个字符串结束标志\0，构成一个字符串。printf 使用 s 格式输出，它们执行是这样的：按字符数组名 line 找到该数组的起始地址，然后逐个输出数组中的字符，直到遇到\0 为止。

例 5.12　把两个字符串连接起来。

```
#define LENGTH 40
main()
{
    char str1[LENGTH + 1], str2[LENGTH + 1] ;
    char result[2 * LENGTH + 1] ;
    int len1, len2 ;
    printf("Input first string. \n") ;
    scanf("%s",str1) ;
    printf("Input second string. \n") ;
    scanf("%s",str2) ;
    len1 = 0 ;
    while (str[len1] != '\0') {
        result[len1] = str1[len1] ;
        len1 ++ ;
    }
    len2 = 0 ;
```

```
    while (str2[len1] ! = '\0') {
        result[len1] = str2[len2] ;
        len1 ++ ;
        len2 ++ ;
    }
    result[len1] = '\0' ;
    printf("%s\n", result) ;
}
```

运行结果

```
Input first string.
Good
Input second string.
bye
Goodbye
```

程序中第一个循环把 str1 的内容送到 result 中，但没有送 '\0'，从第一个字符串的末尾位置开始，第二个循环把 str2 送到 result 中，同样没有送 '\0'，因此在最后我们为新的字符串加一个 '\0' 表示字符串的结束，最后用 printf 输出这个字符串。

例 5.13　将输入的字符串颠倒过来输出。

```
#define MAXLEN 80
main()
{
    char s[MAXLEN + 1], c ;
    int i, j ,len ;
    printf("Input a string. \n") ;
    scanf("%s", s) ;
    len = 0 ;
    while(s[len] ! ='\0')
            len ++ ;
    for (i = 0, j = len - 1; i < j; i ++, j --) {
            c = s[i] ;
            s[i] = s[j] ;
            s[j] = c ;
}
    printf("The string has been reversed:\n") ;
    printf("%s\n",s) ;
}
```

运行结果

```
Input a string.
```

lever

The string has been reversed:

revel

这个程序中的 for 语句中用了逗号表达式,字符串从两端向中间移动。

例 5.16 统计输入的正文中有多少单词。

这里所说的单词是指用空白符分隔开的字符串。C 语言中空白符有空格、制表符和换行符。为了简单一些,我们不考虑一个单词用连字符"-"跨行的情况。

```
#include "stdio.h"
#define YES  1
#define NO 0
main()
{
    int c, n, inword ;
    n = 0 ;
    inword = NO ;
    while ( (c = getchar() ) ! = EOF)
        if (c ==' ' || c ==\t' || c == '\n')
            inword = NO ;
        else if (inword == NO) {
            inword = YES ;
            n ++ ;
        }
    printf("There are %d words\n", n) ;
}
```

运行结果

```
count words in input^ z
There are 4 words
```

本程序中 while 循环每次读入一个字符,直到遇到文件结束标志 EOF 为止。在键盘输入时当打^ z(UNIX 系统下是^ d)时,系统送一个 EOF。循环体中的 if 语句值得研究一下。如当前字符是空白符时,置 inword 标志为 NO,表示不处在一个单词之中。在当前空符不是空白符时,如果 inword 标志为 NO,表明前一个字符是空白符,则当前字符是新单词的开始,单词数加 1,并置 inword 为 YES 表示已处在单词中。如果 inword 不等于 NO(等于 YES),则意味着此字符不是新单词的开始因此不做单词数加 1 的运算。

例 5.15 把一个数字字符串转换为相应的整数。

我们可以把由数字组成的字符串转换成对应的整数值,一个数字字符转为整数可用它的 ASCⅡ码值减去字符 0 的 ASCⅡ码值求得,如

```
i = c - '0' ;
```

程序如下:

```
main()
{
    char s[7] ;
    int i, n, sign ;
    printf("Input a numberic string\n") ;
    scanf("%s",s) ;
    i = 0 ;
    sign = 1 ;
    if (s[i] == '+' || s[i] == '-') /*符号*/
        sign = (s[i ++] == '+') ? 1 : -1 ;
    for (n = 0; s[i] >= '0' && s[i] <= '9'; i ++)
        n = n * 10 + s[i] - '0' ;
    n = sign * n ;
    printf("%d\n", n) ;
}
```

运行结果

Input a numberic string。

567

567

再次运行

Input a numberic string

-123

-123

本程序允许数字串中带有正负号,语句

```
sign = (s[i ++] == '+') ? 1 : -1 ;
```

用到条件运算符,它相当于

```
if (s[i ++] == '+')
    sign = 1 ;
else
    sign = -1 ;
```

第四节　常见错误

1.数组下标越界,如,

```
int a[10], i ;
for (i = 0; i <= 10; i ++)
    scanf("%d", &a[i]) ;
```

由于a定义有10个元素,下标为0~9,当i为10时,实际上scanf形式为

```
scanf('%d",a[10]) ;
```

而数组 a 中根本就没有 a[10]这个元素，所以这次接收输入是错误的。C 语言本身对下标越界不做检查，因此在发生这种错误时，程序可能会继续运行，而把错误带到程序的其他地方。

2. 数组整体赋值。

如

```
int a[10] ,b[10] ;
⋮
b = a ;
```

这是错误的，C 语言不允许对数组作整体的操作，如果想把 a 的值赋给 b，需要用循环来实现，如

```
for (i = 0; i < 10; i ++)
    b[i] = a[i] ;
```

同样也不能用 scanf 一次接收一个数组的值，如

```
scanf("%d", &a) ;
```

是错误的。

3. 对非静态，非外部数组作初始化，如

```
main()
{
    int a[5] = {1,2,3,4,5} ;
    ⋮
}
```

C 语言规定只有静态数组和外部数组才能初始化，所以，定义及初始化应写成

```
static int a [5]={1,2,3,4,5};
```

4. 接收字符串时用了取地址运算符，如

```
char str[20] ;
scanf("%s", &str) ;
```

由于数组名本身就代表地址，所以不应再加 &。实际上，只要是用 s 控制字符，其对应的变量前从不加 &。正确的写法是

```
scanf("%s", str) ;
```

5. 向一个字符数组赋字符串，如

```
char str[20] ;
str="hello" ;
```

这种错误实际上与第二种错误是一样的。C 语言不支持对数组的整体操作，但使用者由于看到字符数组初始化的情形，就以为能够把字符串赋给一个数组，这种错误出现的频率很高，应加以重视。编译会对这种情况给出错误提示。

6. 构造字符串忘记末尾应加\0，如

```
i = 0 ;
while ( ( c = getchar() ) ! = '\n')
    line[i ++] = c ;
printf("%s\n", line) ;
```

由于构造的字符串没有加结束标志，当 printf 输出 line 时，从 line 的起始地址开始一个个地输出字后，输出完读入的字符后没有遇到\0，继续一直输出，这时的内容已不再是字符串中的字符了，可能是乱七八糟一大串，直到在内存中遇到另一处的\0，或访问到不允许访问的地方发生错误才停下来。因此，在自己构造一个字符串时，一定不要忘记在末尾加上\0。

第六章　函　　数

本章学习要点

本章介绍C语言中函数的定义，函数的调用，递归函数以及作用域和存储类的概念。学习本章应掌握如何编写函数，如何利用函数来把较大的问题分解后加以解决，应掌握作用域的概念及各种存储类型变量的使用，进而了解C程序的结构。

在第一章中曾介绍过，C语言是由函数组成的。有了函数，我们可以把一个较大的问题分解，在较高层次上只考虑做什么，而把怎么做放到较低层次考虑。通过函数的定义把实现的细节封闭起来，通过函数调用组合各种功能，就象用一个个构件盖房子似的。这也是结构程序设计的基本要求。

第一节　函数的定义

6.1.1　函数定义的一般形式

我们先来看两个例子。

例 6.1

```
main()
{
    print_star();
    print_message();
    print_star();
}
print_star()
{
    printf("* * * * * * * * * * * * * * * * *\n");
}
print_message()
{
    printf("I like programming in C! \n");
}
```

运行结果

```
* * * * * * * * * * * * * * * * * * *
      I like programming in c
* * * * * * * * * * * * * * * * * * *
```

这个程序定义了三个函数，其中 main 我们早已熟悉，它在 C 程序中是必须的，C 程序总是从 main 处开始执行。函数 print_star 和 print_message 分别完成输出一行星号和输出一行信息的功能，这里定义函数都是无参的，更一般的情况，函数可以带有参数。

例 6.2

```
main()
{
    int i ;
    for (i = 1; i <= 5; i ++)
       fun(i) ;
}
int fun(x)
int x ;
{
    printf("%d\n",x) ;
}
```

运行结果

```
1
2
3
4
5
```

这里函数 fun 有一个参数，当 fun 被调用时，实在参数 i 的值传给 fun 的形式参数 x，在每次调用时输出这个参数值。

现在让我们来看一看函数定义的一般形式

无参函数的定义形式为：

```
类型标识符    函数名()
{
      说明部分
      语句
}
```

有参函数的定义形式为：

```
类型标识符  函数名(形式参数表)
形式参数说明
{
      说明部分
      语句
}
```

类型标识符指出函数的类型，也就是函数返回值的类型，当返回值的类型是 int 或 char 型时可以省略。参数用作在主调函数和被调函数之间传递信息，多个形式参数用逗号分隔，

在函数名后的括号中列出。形式参数的类型说明放在圆括号的后面，但要放在用花括号括起的函数体的前面。如果是无参函数，则没有形参表和形参说明，但函数名后的圆括号不能省略。另外要注意圆括号后不能跟分号。例如

```
int max(a,b)
int a, b ;     /*形参说明*/
{
    int  c ;
    c = (a > b) ? a : b ;
    return(c) ;
}
```

这里定义了一个整型函数 max，这就是说，它返回的函数值为整型。a,b 是形式参数，当函数被调用时，它们的值由主调函数传递过来，它们的说明要写在花括号外面。

一个函数最简单的形式是这样的：

```
dummy()
{
}
```

这是一个空函数，当它被调用时什么也不做。这样的函数在编写程序的过程中常常会用到：在开始写程序的时候，我们只是在主调函数需要调用函数的地方写上一个函数调用语句，在定义的地方写一个空函数，这样可以迅速地构成程序的结构，而不致于过早地陷入实现的细节中。在程序的结构正确地构造好之后，随着程序实现的精细化，再用所需的功能扩展它们。这时改变的只是局部，而不影响程序的整体结构，这种编程方法编出的程序更易于阅读、易于理解，因而也容易保证正确性。

6.1.2 函数的参数

在函数被调用时，一般主调函数和被调函数之间有数据的传递关系，参数的作用就是从主调函数向被调函数传递数据。函数定义中使用的参数叫做形式参数，简称形参，主调函数中对应的参数称做实际参数，简称实参。

例 6.3

```
main()
{
    int x, y, z ;
    printf("Input two numbers \n") ;
    scanf("%d%d", &x,&y) ;
    z = max(x, y) ;
    printf("max is %d\n", z)
}
int max(a, b)
int a, b ;
{
    int c ;
```

```
    c = (a > b) ? a : b ;
    return(c) ;
}
```

运行结果

```
input two numbers
2       5
max is  5
```

在 main 函数中

```
z = max(x,y) ;
```

调用了函数 max，调用时把形式参数 x、y 的值传递给函数 max 对应的形式参数 a、b，如图 6-1 所示。

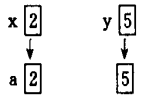


图 6-1　参数传递

形式参数和实在参数应在类型与数目上保持一致，如参数个数不匹配，编译时系统会指出错误。

另外需要强调的是：C 语言的参数都是传值的，也就是说，主调函数把实参值传给形参后，形参的值如何变化都不影响到实参。

例 6.4

```
main()
{
    int a, b ;
    a = 1 ;
    b = 2 ;
    printf("main_1:a=%d,b%d=\n", a, b) ;
    fun(a, b) ;
printf("main_2:a=%d,b=%\n",x,y,);
}
    int fun (x, y)
int x, y ;
{
printf("fun_1:x=%d,y=%d\n", x, y) ;
    x = 8 ;
    y = 7 ;
    printf("fun_2:x=%d,y=%d\n", x, y) ;
}
```

运行结果

main_1:a=1,b=2

fun_1:x=1,y=2

fun_2:x=8,y=7

main_2:a=1,=b=2

从结果可以看到,形参的改变不影响到实参,图 6.2 给出了例 6.4 的执行情况。

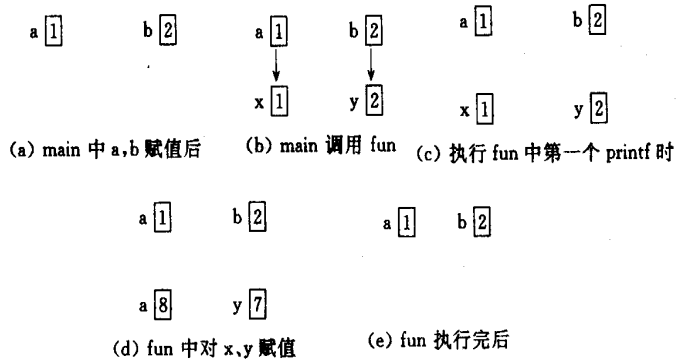


图 6-2 参数的传值效果

从图 6-2 中可以看出,形参和实参各占据不同的存储单元,函数的形参只在函数被调用时才分配有存储单元,函数返回后,形参所占的存储单元自动释放。参数值的传递只在函数被调用时才发生,一但被调函数开始执行,实参与形参的联系就不存在了,因此形参值的任何变化都不会影响到实参。

ANSI C 允许以另一种方式说明形参,即在圆括号中同时给出形参的类型说明,如可用

```
int max(int x, int y)
```

来代替

```
int max(x, y)
int x, y ;
```

这种表示形式确实很好,但考虑到多数 C 程序员的传统习惯,也为了与其他多数讨论 C 语言的书相一致,本书中还是使用形参单独说明的形式。单独说明的形式也可以使你的程序不受具体编译的限制。

6.1.3 函数的返回值和函数类型说明

许多时候,需要函数返回一个值,函数值的返回是用 return 语句实现的。return 语句有两种形式:

return;

或

return 表达式

例如

```
return ;
return -1 ;
return(-1) ;
```

```
return(x+y) ;
return (++x) ;
```

都是正确形式的 return 语句。虽然不是必须的，但使用圆括号将表达式括起显得更清楚些。

return 语句用于结束函数执行，返回主调函数。当 return 后面带有表达式时，将这个表达式的值转换为函数类型说明所指定的类型，并做为函数值返回主调函数。

如果 rnturn 后没有表达式，则只做返回主调函数操作。如果一个不带表达式的 return 语句写在函数的最后，那么它也可以省略不写，函数执行完会自动返回主调函数。

一个函数中可以有多个 return 语句，如，例 6.3 中的 max 函数也可写做

```
int max(x, y)
int x, y ;
{
    if (x > y)
      return(x) ;
    else
      return(y) ;
}
```

如果一个函数有返回值，那么值的类型由函数类型说明决定，如 max 函数的类型为 int 型，则返回的函数值亦为 int 型，C 语言规定：如果函数不明确指出类型，则系统按整型对待，所以 int max(a, b)中的 int 也可以省略不写，但如果函数是其他类型则必须明确写出。

例 6.5

```
main()
{
    float a, b, c ;
    float fadd() ;
    scanf("%f %f", &a, &b) ;
    c = fadd(a, b) ;
    printf("sum is %f\n", c) ;
}
float fadd(x, y)
float x, y ;
{
    return(x + y) ;
}
```

运行结果

```
2.1     3.5
snm is   5.600000
```

这里函数定义部分的 float 是不能省略的，在 main 中，我们说明 fadd 函数为 float 型，有关这一点将在下节做详细讨论。

如果一个函数没有显式地用 return 返回一个值，则函数返回一个不确定的值。为了明确地表示不需要返回值的情况，ANSIC 中引入了关键字 void，用来说明函数没有返回值（实际上许多 UNIX　C 中也引入了 void）。这样就可以对不需要返回值的函数冠以 void，如例 6.1 改写为

```
void print_star()
{
    :
}
void print_message(_)
{
:
}
```

对所有的函数都明确给出类型是个好习惯，即使类型为 int 或 char 也是如此。这样的好处是：你可以清楚地记住任何函数都是有类型的，不至于当类型为 float、double 等时因忘记指出类型而产生错误，而且也使程序更清楚。做为例外，主函数 main 可以不指明类型，当然 main 前加上 int（或 void）也是正确的。

第二节　函数的调用

6.2.1　函数调用的一般形式

函数调用的一般形式为：

函数名（实参表）

如果是调用无参函数，实参表没有，但括号仍要保留。如果有多个实参，则它们之间用逗号分隔，实参应在数目上与形参保持一致。

下面就函数调用作几点说明。

1. 实参表达式求值的顺序是不确定的。这样有时程序会产生意想不到的结果。

例 6.6　求 n^{n-1} 的值。

```
main()
{
    int n, result ;
    printf("n=") ;
    scanf("%d", &n) ;
    result = pow(n, n --) ;
    printf("%d\n", result) ;
}
int pow(x, m)
int x, m ;
{
    int p ;
```

```
    for (p = 1;m > 0; -- m)
        p = p * x ;
    return(p) ;
}
```

先考虑一下当输入值为 4 时输出应是什么值？是 64(4^3)吗？好象应该是。运行一下

```
n=4
result=81
```

这是怎么回事？结果成了 3^4。会不会是 n--的问题？不妨改成--n。这时主函数中

```
result = pow(n, n --) ;
```

改为

```
result = pow(n, -- n);
```

程序的其他部分不变，再执行一次，

```
n=4
result=27
```

结果是 3^3。这种结果产生的原因是实参求值的顺序不是从左到右，而是从右到左。当然你也可以把

```
result = pow(n, n --) ;
```

写成

```
result = pow(-- n, n) ;
```

这样从右到左求实参值时就得到 n^{n-1}。但如果编译是从左到右求值的呢？为了避免这种尴尬，可用

```
result = pow(n, n - 1) ;
```

来求 n^{n-1}。正确的 main 函数如下：

```
main()
{
    int n, result ;
    printf("n=") ;
    scanf("%d", &n) ;
    result = pow(n, n - 1) ;
    printf("result=%d\n", result) ;
}
```

这样不管先求 n 还是先求 n-1，结果都是一样的，因为 n 并没有改变。

前面那种结果难于预测的情况一方面是因为实参求值的顺序各机器不同影响的；另一方面也是增 1 减 1 运算造成的，因为增减 1 运算改变变量自身的值。作为忠告：你不要让一个变量与这个变量自增或自减的运算出现在同一个实参表中。

2. 函数调用相当于一个表达式。

除了明确说明为 void 类型的函数，其他函数总返回一个值，因此，所有可以出现表达式的地方都可以出现非 void 型的函数，这可能有三种情况。

(1)函数以表达式语句的形式出现。

就像赋值表达式能以赋值语句的形式出现一样，函数调用也可以做为语句，如例 6.1 中的

```
print_star() ;
```

这种情况下，void 型函数也是可以的。

(2)出现在表达式中，如

```
c = 2 * max(a, b) ;
```

(3)函数值做为参数。如

```
d = max(max(a, b), c) ;
```

这里 max(a, b)的值为再次调用 max 函数的第一个参数。

3. 调用库函数，有时需要在本文件的前头加上文头件(.h 文件)。

在前面的例子中已经见过，当调用库函数 getchar 和 putchar 时，用到头文件 stdio.h，形式为

```
#include "stdio.h"
```

当调用数学函数，比如说求平方根函数 sqrt 时，用到了 math.h 文件，形式为

```
#include "math.h"
```

在这些.h 文件中存放有使用某些库函数所必要的信息，命令 include 将这些.h 文件的内容包含进本文件之中。关于文件包含的详细讨论将在第九章中进行。

4. 如果一个被调函数不是在本文件中主调函数之前定义，则主调函数调用函数之前应说明被调函数的类型。

在例 6.5 中我们已经见到过这种情况，为了便于阅读，我们再把它抄一遍。

```
main ()
{
    float  a, b, c ;
    float fadd(); /* 函数类型说明 */
    scanf("%f%f", &a, &b) ;
    c = fadd(a ,b) ; /* 函数调用 */
    printf("sum is %f\n", c) ;
}
float fadd(x, y) /* 函数定义 */
float  x, y ;
{
    return(x + y) ;
}
```

由于 main 中调用了函数 fadd，而 fadd 的定义在 main 之后，调用体现为“先使用，后定义”，因此在 main 中调用 fadd 之前说明了

```
float  fadd();
```

如果我们把程序中两个函数的顺序换一下，变成

```
float fadd(x, y)
float  x, y ;
```

```
{
    :
}
main()
{
    :
}
```

则 main 中可以不写

```
float fadd() ;
```

这时体现的是“先定义,后使用”。

我们也可以把函数说明放在 main 函数的前面,写成

```
float  fadd() ;
main()
{
    :
}
float fadd(x, y)
float x, y ;
{
    :
}
```

这里应该指出:函数的定义和函数的说明是不同的,“定义”是指对函数功能的确定,包括指定函数类型,函数名,形参,函数体等,它是一个完整的程序单位,在内存中占据一定的存储空间,而“说明”只是对函数的类型加以指明,以便在主调函数中作相应处理,并不占据独立的内存空间。一个程序中对一个函数只能够定义一次,却可以有多次说明,有关函数的说明问题,在学习了作用域规则(本章第五节)后会有更深的理解。

最后应说明的是:如果是调用整型函数,说明可以省略。

6.2.2 应用举例

例 6.7 求三个整数的最大公因子。

求三个数的最大公因子可以采用这样的方法,先求两个数的最大公因子,然后用这个公因子与第三个数再求最大公因子,这样就得到了三个数的最大公因子。由于要多次求最大公因子,所以最好把它写成一个函数。程序如下:

```
/* 求三个数的最大公因子 */
main()
{
    int a, b, c, g ;
    printf("Input three integers.\n") ;
    scanf("%d%d%d", &a, &b, &c) ;
    g = gcd(a, b) ; /* 求头两个数的最大公因子 */
```

```
    g = gcd(g, c) ; /* 与第三个数再求最大公因子 */
    printf("gcd=%d\n", g) ;
}
/*  用 Euclid 算法求最大公因子 */
int gcd(m, n)
int m, n ;
{
    int r ;
    while (n ! = 0) {
        r = m % n ;
        m = n ;
        n = r ;
    }
    return(m) ;
}
```

运行结果

```
Input three integers.
26  91  65
gcd=13
```

在本程序中，我们把求最大公因子的操作编写成函数，在 main 中两次调用了它。如果不用函数，相同结构的程序段就要重复写两次。在实际中，我们总是倾向于把公共的操作尽可能地写成独立的函数，以便在需要的时候调用，而不是把所有的操作都写到 main 中去。

例 6.8　求 3～100 之间的全部素数。

我们分析这个问题的解时，最直接的想法是：从 3 到 100，如果这个数是素数就输出它。现在我们按这种思路来写程序。

```
/* 求 100 以内的素数 */
#define RANGE 100
main()
{
    int n, i ;
    i = 0 ;
    for (n = 3; n <= RANGE; n ++)
        if (isprime(n) { /* n 是素数 */
            printf("%6d", n) ; /* 输出素数 */
        i ++ ;
        if (i % 5 == 0)
            printf("\n") ;
        }
```

```
    if (i % 5 ! = 0)
        printf("\n") ;
}
```

这个程序以最直接的方式求解素数，程序的功能一目了然，下面的工作就是函数 isprime 了。函数 isprime 很简单，如果 n 能被 2～n－1 间的任何数整除，则它不是素数，返回一个 0 值表示假，否则它是素数，返回 1 表示真。函数 isprime 如下：

```
/ * 判断一个整数是否为素数，如果是素数返回 1，否则返回 0 * /
int isprime(n)
int n ;
{
    int k ;
    for (k = 2; k < n; k ++)
        if (n % k == 0)
            return(0) ;
    return(1) ;
}
```

程序运行结果

```
   3    5    7   11   13
  17   19   23   29   31
  37   41   43   47   53
  59   61   67   71   73
  79   83   89   97
```

从这个例子中可以看出，我们写程序，在较高层次只需要把解题的思路表述清楚，一些细节可用函数调用代替，而不必马上去考虑，在高层描述正确后，再在函数中实现必要的细节。很明显这样编出的程序清晰明了，容易保证正确性，而且编程的效率也高。

例 6.9 求两个实数中较大者

```
main()
{
    float a, b ;
    float fmax() ; / * 函数类型说明 * /
    scanf("%f%f", &a, &b) ;
    printf("max  is%f\n", fmax(a, b) ) ;
}
float fmax(x, y)  / * 函数定义 * /
float x, y ;
{
    return( (x > y) ? x : y) ;
}
```

这个程序中 main 调用了浮点型函数 fmax，而 fmax 的定义在 main 之后，所以在 main 中说明了 fmax 的类型，当我们习惯直接调用整型函数时，不要忘记，其他类型函数需要说明。

6.2.3 函数的嵌套调用

C 语言中所有函数的定义都是平行的，也就是说，不能象 PASCAL 那样在函数定义中再定义其他函数，但 C 语言允许在函数定义中再调用其他函数，比如说函数 a 调用函数 b，而函数 b 又调用函数 c，这就是函数的嵌套调用。

例 9.10　用牛顿迭代法求一个正实数的平方根。

牛顿迭代法是用计算机求解平方根近似值的最简单的方法之一。其求值过程如下：

1°　设置猜测初值为 1。

2°　如果|(猜测值)^2−x|<ε，则转 4°

3°　置新猜测值为(x/猜测值+猜测值)/2，返回 2°

4°　猜测值是满足精度要求的 x 的平方根。

程序如下：

```
/* 求一个实数的平方根 */
main()
{
    float squ_rt(), a ;
    printf("Input a=") ;
    scanf("%f", &a) ;
    if (a < 0)
        printf("Negative argument to square root.\n") ;
    else
        printf("square_root(%f)=%f\n", a, squ_rt(a) ) ;
}
/* 用牛顿迭代法求 x 的平方根 */
float squ_rt(x)
float x ;
{
    float abs_value( ) ;
    float epsilon, guess ;
    epsilon = 1E - 5 ;
    guess = 1.0 ;
    while (abs_value(guess * guess - x) >= epsilon)
        guess = (x / guess + guess) / 2.0 ;
    return(guess) ;
}
/* 求 x 的绝对值 */
```

```
float abs_value(x)
float x ;
{
    if (x < 0)
        x = -x ;
    return(x) ;
}
```

执行结果

```
Input  a=2.0
square_root(2.000000)=1.414216
```

在本程序中，函数 main 调用了求平方根的函数 squ_rt，而 squ_rt 函数中又调用了函数 abs_value，这种函数调用套函数调用的结构如图 6-3 所示。

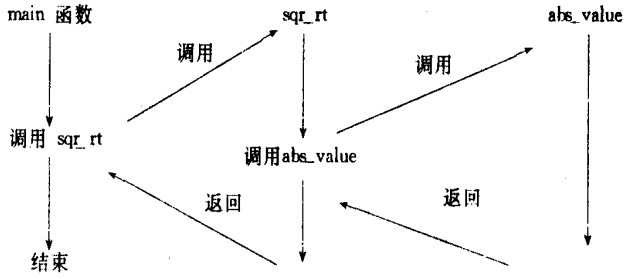


图6-3　函数的嵌套调用

函数的嵌套调用使程序形成了一种自顶向下树形结构，如图 6-4 所示。

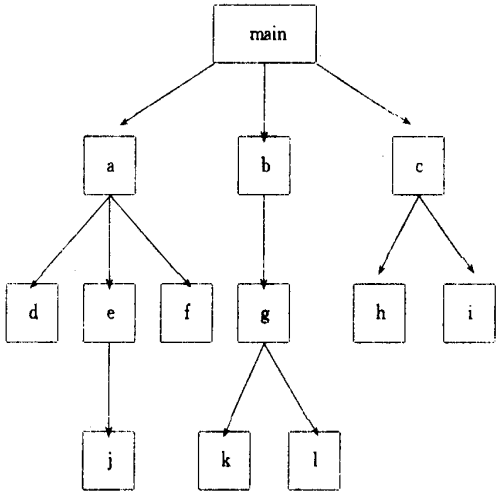


图6-4　程序的树型结构

也就是说，C 程序总是从 main 开始执行，在 main 中往往需要调用其他函数，而这些函数又可以调用另一些函数。通过这样的形式，我们可以用自顶向下，逐步求精的方法编写程序：在高层只考虑做什么，在需要的地方写一个函数调用，而把函数实现放到以后，同样在定义一个被调函数时也采用同样的办法，把更细的实现放到更下层的函数中，直到一个函数只完成

简单、单一的功能为止。这种结构的程序更易于阅读理解，更容易保证正确性，也更容易维护和修改。一个C程序通常不是由少数几个大函数，而是由许许多多小函数组成，一般一个函数只有几十行甚至只有几行。

第三节　数组作为函数参数

在前面的例子中，我们看到单个变量可以做函数的参数，其实数组也可以做为函数的参数。

例6.11　顺序查找。

查找是计算机中经常要遇到的一种操作，其含义是在一组数据中查找到所查数据的位置。当一组数据无序时，一般采用顺序查找。顺序查找是把给定的值与这组数据中的每个值顺序比较，如果找到，就输出这个值的位置，如果找不到则报告没找到。下面是顺序查找的程序。

```
/*在一组数据中查找给定数据的位置*/
#define SIZE 10
main()
{
    int d[SIZE], key, i, index ;
    printf("Input %d numbers\n", SIZE) ;
    for (i = 0;i < SIZE ;i ++)
        scanf("%d", &d[i]) ;
    printf("Input a key you want to search\n") ;
    scanf("%d", &key) ;
    index = seq_search(d, SIZE, key) ;
    if (index >= 0)
        printf("the index of the key is%d\n", index) ;
    else
        printf("Not found.\n") ;
}
/*求给定值key的位置，找到则返回其下标，找不到返回-1*/
int seq_search(v, n, key)
int v[] ; /*数组参数*/
int n ; /*数组元素个数*/
int key ; /*待查的值*/
{
    int i ;
    for (i = 0; i < n; i ++)
        if (key == v[i])
            return(i) ;
```

```
    return(-1) ; /* 没找到 */
}
```

运行结果

```
Input  10  numbers.
9   5   7   11   2   3   0   10   8   1
Input  a  key  you want to search
3
The index of the key is 5
```

本程序中使用了一个数组做为参数，函数 seq_serach 中形参 v 没有指定数组的长度，要处理的元素的个数由其后所跟的参数 n 决定，当然也可以把形参 v 说明为

```
int  v[SIZE] ;
```

但这样在实际中并不方便，它使函数 seq_serach 依赖于具体问题，失去了通用性。所以最好是按程序中那样写。

我们在前面讲过，C 语言的参数都是“传值的”，那么数组名参数代表什么值呢？C 语言规定，一个数组名代表存放那个数组的内存首地址，所以它实际上是一个地址值。

传递一个地址值对程序会造成什么影响呢？假定数组 d 的首地址是 2000，则 d 的各元素值顺序地存放在 2000 开始的存储单元中，如图 6-5 所示。

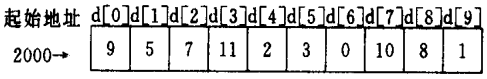


图 6-5　数组 d 的存储形式

当函数 seq_serach 被调用时，将实参 d 的首地址传给形参 v，于是数组 v 的首地址也变为 2000，各元素放置在以 2000 开始的地址空间中，如图 6-6 所示。

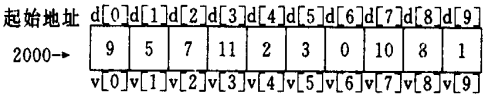


图 6-6　数组 v 的存储形式

也就是说形参数组元素与对应的实参数组元素占有相同的存储单元。这样一来，形参数组中元素值的改变就都会反过来影响到实参数组。

关于这一点，我们来看一看下面的程序。

例 6.12　shell 排序。

shell 排序是以发明者命名的一种较快的排序方法，我们来看一下它的操作过程。现有一组数据

8　3　4　2　9　5　1　7　0　6

首先我们把它们相隔 5 个作比较如前面的大于后面的，就将两者对换，比如说 8 与 5 比较，8>5，则两者互换，再比较 3 与 1，3>1，进行互换，比较 4 与 7，4<7，不需要互换，这样一遍扫描后，数据顺序为

5　1　4　0　6　8　3　7　2　9

然后把这些数相隔 3 个比较，第二遍扫描后的结果为

0　1　4　3　6　2　5　7　8　9

再以增量 2,1 进行比较,最后得到

0 1 2 3 4 5 6 7 8 9

增量序列是可以改变的,唯一的原则是最后一个增量必须是 1。取什么样的增量序列最好,是数学上非常复杂的问题,但大量测试表明:9,5,3,2,1 是一个效果较好的序列。如果为了简单,我们也可以用每次减半的方法得到一个增量序列,下面的程序就采用这种方法。

```
/*将输入的数据排序后输出。一个测试 shell 排序的主函数*/
#define SIZE 10
main()
{
    void shell( ) ;
    int d[SIZE], i ;
    printf("Input  %d  numbers\n", SIZE) ;
    for (i = 0;i < SIZE; i ++)
        scanf("%d", &d[i]) ;
    shell(d, SIZE) ;
    printf("After  sort:\n") ;
    for (i = 0; i < SIZE; i ++)
        printf("%5d", d[i]) ;
    printf("\n") ;
}
/*把数组 v 的元素按增序排列*/
void shell(v, n)
int v[], n ;
{
    int gap, i, j, temp ;
    for (gap = n / 2; gap > 0; gap /= 2) /*扫描次数*/
        for (i = gap; i < n; i ++) /*一遍扫描*/
            for (j = i - gap; j >= 0 && v[j] > v[j + gap]; j -= gap) {
                temp = v[j] ;
                v[j] = v[j + gap] ;
                v[j + gap] = temp ;
            }
}
```

运行结果

```
Input  10  numbers
8   3   4   2   9   5   1   7   0   6
After  sort:
0   1   2   3   4   5   6   7   8   9
```

从结果中我们看到:当做为形参的数组 v 的元素改变后,也影响到了主调函数中的实参

数组 d 的元素。这与 C 语言的参数都是传值的有矛盾吗？没有！作为参数值的地址值并没有改变，改变的只是地址中存放的内容。通过这种方式，我们可以间接地改变主调函数中的数据。

如果一组数据已经排好序，则查找时可以采用较快的方法，比如说折半查找。

例 6.13 折半查找。

折半查找的思路是：先查找一组数据中间的元素，如果是要查找的数据，查找成功，否则如果要查找的数比中间的数据值大，说明要查找的数在后半部，就在后半部分查找，如果要查找的数比中间元素的数值小，说明要查找的数应在中间值的前面，就应继续在前半部分查找，这样查找的范围就缩小了一半，然后在这半部分中再用折半的方法继续查找，直到找到，这时给出数值所在的位置，或者最终找不到，这时给出找不到的信息。这里我们只给出折半查找本身。

```
int binary(v, n, key)
int v[], n, key ;
{
    int low, high, mid ;
    low = 0 ;
    high = n - 1 ;
    while (low <= high) {
        mid = (low + high) / 2 ; /* 折半 */
        if (key < v[mid])
            high = mid - 1 ; /* 在前半部分查找 */
        else if (key > v[mid])
            low = mid + 1 ; /* 在后半部查找 */
        else /* 找到 */
            return(mid) ;
    }
    return(-1) ; /* 没找到 */
}
```

单独这样一个函数是不能上机运行的，如要上机运行，需要加上一个测试用的 main 函数。你可以用例 6.11 的 main 函数对 binary 做测试，只需要将语句

```
index = seq_search(d, SIZE, key) ;
```

改成

```
index = binary(d, SIZE, key) ;
```

即可，运行结果与例 6.11 会是一样的。

C 语言中使用得最多的数组是字符数组，下面我们来看一下字符数组做参数的情况。

例 6.14 把一个字符串反序后输出。

```
#define LENGTH 80
main()
```

```
{
    void reverse() ;
    char str[LENGTH + 1] ;
    scanf("%s", str) ;
    reverse(str) ;
    printf("%s\n", str) ;
}
/* 反序一个字符串 */
void reverse(s)
char s[] ;
{
    int c, i, j ;
    for (i = 0, j = strlen(s) - 1; i < j; i ++, j --) {
        c = s[i] ;
        s[i] = s[j] ;
        s[j] = c ;
    }
}
```

运行结果

```
abcdefg
gfedcba
```

这个程序中，函数 reverse 没有指定数组元素个数的参数，这是因为字符串总是以一个\0 结束，通过\0 就可以得到字符数组中元素的个数，也就是字符串的长度，这里我们可以看到使用字符串结束标志的好处，函数 strlen 是一个库函数，它给出字符串的长度。

字符数值做为函数参数在 C 语言中使用得很频繁。下面给出一些例子。希望它们能有助于你的学习。这些例子只给出函数。如果你要运行它们，需要给出 main 函数，并在 main 中调用它们。

例 6.15 求字符串的长度

这是前面用的库函数 strlen 的一个版本。

```
/* 返回 s 的长度 */
int str_len(s)
char s[] ;
{
    int, i ;
    i -- 0 ;
    while (s[i] ! = '\0')
        i ++ ;
    return(i) ;
```

```
}
```

例 6.16

```
/* 复制 t 的内容到 s */
void str_cpy(s,t)
char s[], t[] ;
{
    int i ;
    i = 0 ;
    while ( (s[i] = t[i]) ! = '\0')
        i ++ ;
}
```

这个函数把字符串 t 的内容拷贝到 s 中。一般在操作系统中拷贝命令通常是

```
cp  from  to
```

也就是把第一个文件复制到第二个文件中。而 str_cpy 采用的方式如同赋值

```
s = t
```

是把第二个参数复制到第一个中(C 语言不允许字符串直接赋值)。程序执行时每次把 s 的一个元素送给 t,当送完\0 时循环结束,这时将 s 中的\0 一起复制到了 t,使 t 仍然是一个字符串。

例 6.17

```
/* 把字符串 t 连结到字符串 s 的尾部,版本 1 */
void str_cat(s,t)
char s[], t[] ;
{
    int i, j ;
    i = j = 0 ;
    while(s[i] ! ='\0') /* 找 s 的尾部 */
        i ++ ;
    while( (s[i] = t[j]) ! = '\0') { /* 复制 t 到 s 后 */
        i ++ ;
        j ++ ;
    }
}
```

第一个 while 循环找到 s 的结束标志\0,第二个 while 循环从\0 的位置开始复制 t 的内容,原来\0 的位置就被覆盖了,最后 t 的全部内容(包括\0)复制到了 s 的后面,也就是把 s 与 t 连结起来,结果放在 s 中,很明显,你应该保证 s 有足够的空间。

C 语言常倾向于把程序写得简单,语句

```
i ++ ;
```

```
    j ++ ;
```

也可以与数组元素赋值写在一起，下面的形式在C程序中更常见。

```
    /* 连结 t 到 s 尾部，版本 2，必须保证 s 有足够的空间存放 t */
    str_cat(s,t)
    char s[], t[] ;
    {
        int i, j ;
        i = j = 0 ;
        while(s[i] ! = '\0')
            i ++ ;
        while([(s[i ++] = t[j ++]) ! = '\0')
            ;
    }
```

由于增1运算已写到while行中，所以循环体为空。

你可能会问：第一个循环中的i++能不能写进while行呢？由于条件是判断s[i]是否为\0，只有当s[i]不是\0时才做i增加1，而要是写成

```
    while s[i ++] ! = '\0')
        ;
```

则在s[i]等于\0后i仍然增加了1，也就是把下标置在了\0之后，这在后面复制时，\0就不会被覆盖，而是把t复制到\0后，由于字符以\0结束，后面的内容就不再有意义了。

例6.18 字符串比较

字符串也可以比较大小，也就是按它们对应的字符在ASCⅡ中的值比较。如果字符串中出现的字符都是字母，比较小的字符串就相当于排在字典的前面，比较大的字符串相当于排在字典的后面，所以字符串比较大小一般说成是按字典顺序比较，程序如下。

```
    /* 按字典顺序比较 s 和 t，如果 s<t，返回<0 的值，如果 s=t 返回 0 值，如果 s>t 返回
    大于 0 的值 */
    int str_cmp(s,t)
    char s[], t[] ;
    {
        int i ;
        i = 0 ;
        while (s[i] == t[i])
            if (s[i ++] == '\0')
                return(0) ;
        return (s[i] - t[i]) ;
    }
```

本函数逐个比较s[i]和t[i]的值，如果两个字符串中的每个对应字符都相等，同时到达结束标志\0，则返回0值表示两个字符串相等。一但遇到一对不等的字符，就结束循环，返回

两个字符的差值，如 s[i]的 ASC Ⅱ 值小于 t[i]的 ASC Ⅱ 值，就返回负数，表示 s<t，反之返回正值表示 s>t。

程序中字符位置的移动在 if 语句中实现，

```
if (s[i ++] == '\0')
```

先判断 s[i]是否为\0，然后 i 增加 1，再次判断循环条件时，s 和 t 都移到了下一个字符的位置上。如果不用这种形式，则循环可写成

```
while (s[i] == t[i]) {
    if (s[i] == '\0')
        return(0) ;
        i ++ ;
}
```

这种形式也许更好懂，但 C 程序总是倾向于写得简练些，因此前面的写法更具有 C 的特色。

例 6.19

```
/ * 返回 t 在 s 中的位置，如果 t 不在 s 中返回 -1 * /
int str_index(s,t)
char s[], t[] ;
{
    int i, j, k ;
    for (i = 0; s[i] ! = '\0'; i ++) {
    for (j = i, k = 0; t[k] ! = '\0' && s[i] == t[k]; j ++; k ++)
        ;
    if (t[k] == '\0')
        return(i) ;
    }
        return(-1) ;
}
```

外层 for 循环确定字符串比较的起点，内层 for 循环则是从 s 的某个位置开始，与子串 t 作比较，如果把内层 for 的 j 从 0 开始，与例 6.18 是很相像的。据此，你可以试着分析一下这个函数的实现。

第四节 递　　归

在一个函数中直接或间接地调用该函数自身称做函数的递归调用。C 语言支持函数的递归调用。

递归的概念在我们的自然生活中并不陌生。中国人大多都知道一个古老而有趣的童谣：从前有座山，山里有座庙，庙里有个老和尚讲故事，讲的是：从前有座山，山里有座庙……。这讲的故事又是其自身，这就相当于递归。你也许还见过这样一幅产品包装，上面画着一只猫，猫的手里又拿着产品本身，当然你看到的猫拿的产品包装上面也有只猫拿着产品……这也是递归的例子。在数学中，更是有许多函数采用递归的定义形式，如前面介绍过的菲波那契

数列

$$F_n=\begin{cases}0 & n=0\\ 1 & n=1\\ F_{n-2}+F_{n-1} & n>1\end{cases}$$

当 n>1 时，定义 F_n 用到了前两个菲波那契数。一个更常见的函数是阶乘，它定义为

$$n!=\begin{cases}1 & n=0\\ n\cdot(n-1) & n>0\end{cases}$$

下面我们就来编一个求阶乘的程序。

例 6.20

```
main()
{
    long fact() ;
    int i ;
    for (i = 0; i < 11; i ++)
        printf("%2d! =%ld\n", i, fact[i]) ;
}
/* 用递归方法求 n 的阶乘 */
long fact(n)
int n ;
{
    long result ;
    if (n == 0)
        result = 1 ;
    else
        result = n * fact(n - 1) ; /* 递归调用 */
    return(result) ;
}
```

运行结果

```
0! =1
1! =1
2! =2
3! =6
4! =24
5! =120
6! =720
7! =6040
8! =40320
9! =362880
10! =3628800
```

函数 fact 包含了对其自身的调用，因此 fact 是一个递归函数。现在让我们通过求 3! 来看一看这个递归函数的执行过程。

由于 3≠0，所以执行 else 后的语句

```
result = n * fact(n - 1) ;
```

也就是

```
result = 3 * fact(2) ;
```

这里再次调用了 fact，参数值为 2，由于 2≠0，故应执行

```
result = 2 * fact(1) ;
```

还是要调用 fact 函数，这时参数值为 1，由于 1≠0，继续执行 else 后的语句

```
result = 1 * fact(0) ;
```

这次调用 fact 时，参数值为 0，所以执行

```
result = 1 ;
```

然后将 result 的值返回调用处，这样上一次的调用语句

```
result = 1 * fact(0) ;
```

就变成了

```
result = 1 * 1 ;
```

计算出 result 后，将其值，也就是 fact(1)的值返回调用处，于是

```
result = 2 * fact(1) ;
```

成为

```
result = 2 * 1 ;
```

将其值做为 fact(2)的值返回调用处，

```
result = 3 * fact(2) ;
```

变为

```
result = 3 * 2 ;
```

这样最终得到了 fact(3)的值 6，并将其返回主函数 main 中输出。归纳起来 fact(3)的求值过程相当于

```
fact(3)=3*fact(2)
       =3*2*fact(1)
       =3*2*1*fact(0)
       =3*2*1*1
       =6
```

图 6-7 给出了求 fact(3)时函数的调用情况。

虽然递归函数的执行过程很复杂，但你没有必要去关心。而用递归的方法描述问题却很简单，程序也更容易读。问题是递归执行的速度很慢(你可以从执行的复杂程度感觉到)。所以象阶乘这样明显可用递推形式求解的问题，最好不要用递归。下面用递推形式重写函数 fact。

```
/*用递推形式求 n 阶乘*/
long fact(n)
int n ;
```

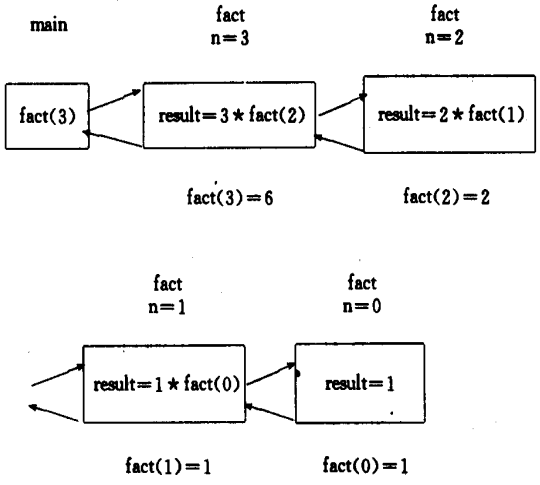


图6-7 递归求解3!

```
{
    long result ;
    int i ;
    result = 1 ;
    for (i = 1; i <= n; i ++)
        result = i * result ;
    return(result) ;
}
```

这种形式同样易于理解,可是执行要快得多。

并不是所有问题都可用递推方式解决。我们再来看一个例子。

例 6.21 汉诺(Hanoi)塔问题。

关于这个问题有一个有趣的故事,在十九世纪末,欧洲流行着一种叫汉诺塔的游戏,这个游戏附有一份推销材料,上面说,布拉玛神庙的教士当时正在玩这个游戏,他们游戏的结束标志着世界的末日。游戏的装置是有三根针,针上从小到大放有 64 个盘子,(市场上销售的则是有 8 个盘子,如图 6.8 所示),游戏的目标是把左边针上的塔移到右边针上,条件是一次只能移动一个盘子,并且不允许大盘放在小盘上面。

先考虑一下这个游戏怎么玩? 如果只有一个盘子,很简单,直接把盘子拿到 C 上,如果是两个盘子呢? 也不难,先把 A 上的小盘子拿到 B,再将 A 上的大盘子放到 C,最后把 B 上的盘子放到 C 上,顺序是

A→B,A→C,B→C

现在考虑一下三个盘子,首先把 A 上最小的盘子放到 C,接着把 A 上中间的盘子放到 B,再把 C 上的盘子放到 B,这时可以把 A 上的大盘子放到 C,然后把 B 上的小盘子放到 A,接着把 B 上的中盘子放到 C,最后把 A 针上的盘子放到 C 上,顺序为

A→C,A→B,C→B,A→C,B→A,B→C,A→C

那么 4 个盘子呢? 问题越来越复杂,我简直想像不出 64 个盘子时该怎么办?

```
    void movetower() ;
    printf("Input the number of disks:") ;
    scanf("%d", &n) ;
    movetower(n, 'A', 'B', 'C') ;
}
/*移动塔*/
void movetower(n,a,b,c)
int n ;
char a, b, c ;
{
    void movedisk( ) ;
    if (n > 0) {
        movetower(n - 1, a, c, b) ;
        movedisk(a, c) ;
        movetower(n-1, b, a, c)
    }
}
/*移动一个盘子*/
void movedisk(from, to)
char from, to ;
{
    printf("%c→%c\n", from, to) ;
}
```

运行结果

```
Input the number of disks:3
A→C
A→B
C→B
A→C
B→A
B→C
A→C
```

最后必须提醒注意一点:编写递归函数时,必须在函数的某些地方使用 if 语句,强迫函数在没有执行递归调用之前就返回,如不这样做,一但调用递归函数,将永远不会返回。

第五节　存储类和作用域规则

在 C 语言中每一个变量或函数都具有两个属性:类型和存储类。类型规定了它们的取值范围和可参与的运算,存储类则规定它们以何种方式存储,以及它们在什么范围内是可见

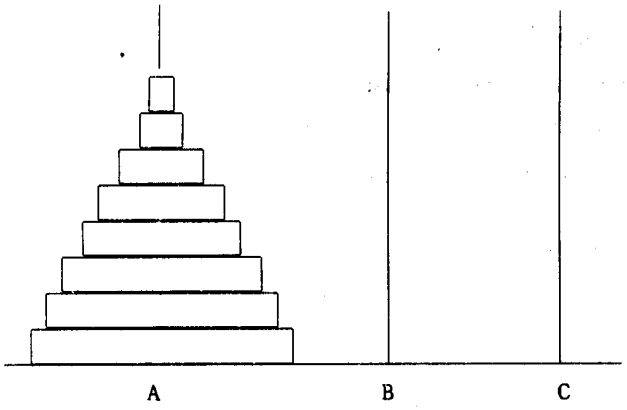


图6-8 汉诺塔

现在让我们来换一种思路:如果能一次移动63个盘子,则移动64个盘子的问题就好解决了:

将63个盘子从A移到B

将最后一个盘子从A移到C

将63个盘子以B移到C

这实际上与只有两个盘子情形类似。以这种思路,一个求解

movetower(64,A,C)

的问题可分解为

movetower(63,A,B)

movedisk(A,C)

movetower(63,A,C)

问题还没有解决,但已看到了曙光,我们可以把移动63个盘子问题分解为移动62个盘子的问题……,依此类推,最后我们可分解为只移动一个盘子。这种方法简单而有效,说它简单是我们只需要把一个操作分解为三步,而且从直觉上就知道它是正确的,说它有效,是我们每一次分解都朝最终目标前进了一步,这最终可导致问题的彻底解决。剩下的问题就只在于是否允许递归了。为了构造一个一般算法,我们需要明确指出哪一根针用于存放临时的塔,于是表示方法可用

movetower(n,A,B,C)

它表示,把A针上的n个盘子移到C针上,用B针存放临时塔,它可分解movetower(n−1,A,C,B)

movedisk(A,B)

movetower(n−1,B,A,C)

显然当n<1时不能分解,而当n=1可以直接移动一个盘子,所以当n<1时根本就不再需要做什么。这样我们可以得到求解汉诺塔问题的程序。

```
/* 求 Hanai 塔问题 */
main()
{
    int n ;
```

的，即所谓作用域。

C 语言的存储类分为四种，它们是

auto	自动的
extern	外部的
static	静态的
register	寄存器的

6.5.1 分程序和作用域规则

在讨论存储类之前，我们先来介绍一下分程序的概念。

分程序是由一对花括号括起来的一段程序，这样，一个函数体就是一个分程序，一个复合语句也是一个分程序。C 语言中分程序结构允许并列或嵌套。函数可以看成是最外一层并列的分程序，只是它被命名，并可以带参数和具有返回值。

分程序的重要性在于在每个分程序中都可以定义变量，并且限制变量的作用域。这也就是说变量的说明可以跟在任何复合语句的左花括号后面，而不仅仅限于出现在函数开始的左花括号后面。在分程序中说明的变量其作用域到分程序结构的右花括号为止，也就是说变量的作用域是说明它的分程序。例如：

```
if (n > 0) {
    int i ;
    for (i = 0; i < n; i ++)
        printf("%d\n", i) ;
}
```

变量 i 只在 if 语句条件为真这段语句中有效。如果一个分程序外部也定义了一个同名变量怎么样呢？我们来看一个例子。

例 6.22

```
main()
{
    int i, j ;
    i = 1 ;
    j = 5 ;
    printf("%d,%d\n", i, j) ;
    {
        int i = 8 ; /* 又定义了一个 i */
        i ++ ; /* 内层 i */
        j ++ ; /* j 还是外层的 */
        printf("%d,%d\n", i, j) ;
    }
    i ++ ; /* 外层 i */
    j ++ ;
    printf("%d,%d\n", i, j) ;
}
```

运行结果

1,5

9,6

2,7

从运行结果中可以看出，内层分程序中定义的 i 只在内层分程序中有效，(在内层输出了它的值 9)；而对外层的 i 没有影响。在外层分程序中定义的 j，由于内层分程序中没有重新定义，则它在内层分程序中也是有效的(输出值 6)并影响到后面。而外层中定义的 i，由于在内层分程序中又重新定义了 i，则它在内层分程序中无效，而从内层分程序中出来后，它继续有效(输出的值为 2)。实际上内层的 i 和外层的 i 在计算机内存中占据不同的存储单元，系统把它们看做是不同变量。

关于上面变量名的讨论，也可以扩展到所有名字(标识符)。C 语言的分程序作用域规则规定：每个分程序可有自己的命名，外层分程序说明的名字，如在内层分程序中没有重新定义，那么它在内层分程序中继续有效，如果内层分程序重新定义了这个名字，则外层的名字就被隐藏起来，内层分程序中有效的是重新定义的名字，退出内层分程序后，外层定义的名字继续有效。

虽然 C 语言允许在所有分程序中定义名字，而且有时也很方便，但在实际中，你几乎看不到在一般复合语句中的定义，人们总是倾向于只把定义部分写在函数的开始部分。C 语言中所有函数都是并列的，在一个函数中定义的名字，其作用域是这个函数，而在这个函数外，它们是不可见的。

如

```
f1(a, b)
int a, b ;
{
    int c, d ;        } 此范围内 a,b,c,d 有效但 a 与 f2 中的 a 没有关系
      :
}
f2(m, n)
int m, n ;
{
    int a ;           } 此范围内 m,n,a 有效，但 a 与 f1 中的 a 没有关系
      :
}
main()
{
    int i, j, k ;     } 此范围内 i,j,k 有效
      :
}
```

从上面的定义中可以看到，函数中定义的变量只在本函数内有效，不同函数中变量可以用相同的名字，如 f1 和 f2 中的 a，但它们仍代表不同的变量。

6.5.2 自动变量

C语言中自动存储变量使用得最多。在我们前面的例子中，除讨论数组初始化时定义过的带有static的数组外，其余变量都是自动的。C语言规定，在函数(或分程序)内定义的名字只要不加存储类说明，都表示自动的。当然你也可以不用缺省形式，显式地写出auto。例如：

```
main()
{
    auto int i, j ;
    auto float a, b ;
    auto char c ;
          :
}
```

它就等价于

```
main()
{
    int i, j ;
    float a, b ;
    char c ;
        :
}
```

存储类说明auto一般被省略。

自动变量的存储空间是这样分配的：当进入一个函数(或分程序)时，系统自动地为该函数(或分程序)定义的自动变量分配存储空间。这样，在这个函数(或分程序)中，这些变量是可访问的，当函数(或分程序)执行完毕后，自动变量所占的存储空间被系统自动回收，或者说被自动释放，因而这些变量就不再存在，下次再调用该函数(或分程序)时，系统再为这些变量分配存储空间。正是因为这种变量随函数(或分程序)的运行而产生，随函数(或分程序)的执行完毕而消失才把它们叫做自动变量。

6.5.3 外部变量

任何在函数外部定义的变量都是外部的。此外，外部存储类也适用于函数，C语言规定所有函数都是外部的。这也就是说，函数只能定义在其他任何函数之外，而不允许函数定义中再出现函数定义。

外部变量的作用域可以是整个程序，一般来讲，如果没有特殊说明，它的作用域是从定义处到本文件结束。在函数外说明了某些变量后，后面所有函数都可以对它们进行访问，外部变量的值在整个程序运行期间一直保存。

例6.23

```
int x = 1 ;
main()
{
    void f();
```

```
    printf("%d\n", ++i);
    f();
}
void f()
{
    printf("%\n", ++i);
}
```

运行结果

```
2
3
```

由于i在所有函数之外定义,因此它是一个外部变量。在主函数main中引用了i,先把它加1,然后输出值2,在函数f中又一次访问它,再做加1运算后输出值3。这里,我们可以看出,外部变量的作用域不限于一个函数,在任何一个函数中改变变量的值,都将影响到其后使用它的函数。

我们也可以在引用外部变量的函数中再使用存储类说明符extern来说明。如:

```
int i = 1;
main()
{
    void f()
    extern int i;
    printf("%d\n", ++i);
    f();
}
void f()
{
    extern int i;
    printf("%d\n", ++i);
}
```

存储类标识符extern告诉系统,变量类型和名字已在别处定义过了,这里的

```
extern int i;
```

只是说明一下i是外部变量,如果外部变量的定义在使用之前,不用extern说明也可,但如果定义在使用后面,说明就是不可缺少的了(就象非整型函数)。如

```
main()
{
    printf("%d\n", i);
}
int i = 1;
```

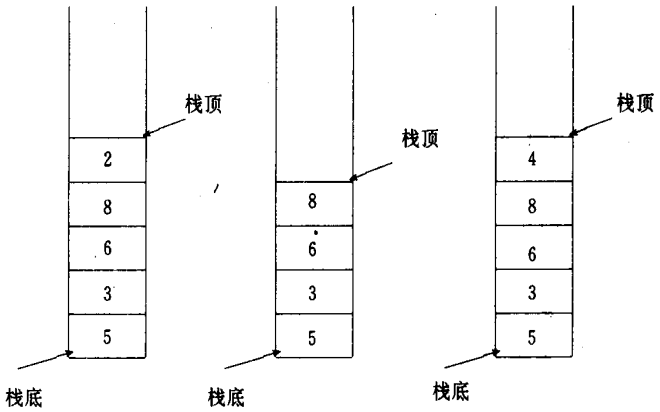
在编译时,系统会毫不客气地指出i没有定义。这时你就需要把程序写成

```
main()
```

```
{
    extern int i ;
    printf("%d\n", i) ;
}
int i = 1 ;
```

运行结果为 1。

外部变量在几个函数没有调用关系而又需要共同操作某些数据时非常有用。考虑一下栈的问题。所谓栈是一种后进先出的结构，我们把一个元素放入栈中时，只能放到栈顶(称压栈)，每次取一个元素时只能从栈顶取(称出栈)。就象一叠盘子，每次放盘子总放到最上面，每次拿盘子时也从最上面拿，也就是说最后放上的最先拿，最先放上的最后拿。栈的结构如图 6-9 所示。



(a)栈　　(b)栈顶元素出栈　　(c)压栈

图6-9　栈的结构

从图中可以看到，不管是压栈还是出栈，都要对同一个数据结构栈进行操作，而且栈顶指针 sp 也是公共的，这种多个函数需要使用共同数据的情况，采用外部变量是比较方便的。如：

```
#define SIZE 100
int sp = 0 ; /* 指向栈顶的指针 */
char stack[SIZE] ; /* 栈 */
void push(i) /* 压栈 */
{
    ⋮
}
int pop() /* 出栈 */
{
    ⋮
}
```

这样在 push 和 pop 中都可以用到 stack 和 sp，而且由于 push 和 pop 操作于同一个栈，也不能够让它们具有自己的栈和栈顶指针。

如果不同 extern 说明，外部变量的作用域是从定义处到本文件的结束，对于定义前的程序部分是不可见的。利用这一点，我们可以把一些不需要由其他函数知道的细节隐藏起来。比如说利用栈实现字符串的倒置。

例 6.24 用栈将字符串倒置。

我们每取出一个字符就把它压栈，当整个字符串中的字符都压栈后，再把它们一个个从栈中弹出，利用栈后进先出的特点进行顺序倒置。现在的问题是，在字符压栈和弹出时，可以不必关心栈的结构和栈顶位置，所以，我们能把有关栈的全部内容，包括外部变量的定义和栈操作函数都放在字符串倒置操作之后，使其数据结构对字符串倒置是不可见的。我们的程序可以写做

```
#define LENGTH 80
main()
{
    void push() ;
    char pop() ;
    char str[LENGTH + 1] ;
    int i, j ;
    scanf("%s", str) ;
    i = 0 ;
    while (str[i] ! = '\0')
        push(str[i ++]) ;
    for (j = 0; j < i; j ++)
        s[j] = pop() ;
    printf("%s\n", s+r) ;
}
#define SIZE 100 / * 栈的最大尺寸 * /
int sp = 0 ; / * 指向栈顶的指针 * /
char stack[SIZE] ; / * 字符栈 * /
void push(i) / * 压栈运算 * /
char i;
{
    if (sp < SIZE)
        stack[sp ++] = i ;
    else { / * 栈满 * /
        printf("error:stackfull. \n") ;
        clear();
    }
}
char pop() / * 弹出栈顶元素 * /
{
```

```
    if(sp > 0)
        return(stack[-- sp]) ;
    else {
        printf("error:stack empty.\n") ;
        clear() ;
    }
}
void clear() /* 清空栈 */
{
    sp = 0 ;
}
```

这里 push 和 pop 操作同样的数据结构，都用到了栈 stack 和栈顶指针 sp，而且一个函数的操作必须影响到其他函数，而它们之间又没有调用关系，所以使用外部变量是再自然不过的事儿了。

如果已定义了一个外部变量，而在某个函数中又定义了一个同名的变量会怎么样呢？与程序中的自动变量类似，在这个函数中使用它自己定义的变量，外部变量被暂时掩盖起来，如

```
int x = 10 ;
main()
{
    int x = 5 ;
    printf("%d\n", x) ;
}
```

将输出 5 而不是 10。

一般来讲，一个 C 程序可以放在几个文件里，这样一来，外部变量的定义和使用它们的函数就有可能出现在不同的文件中，在组成程序的所有文件中，必须且只能有一个定义，这个定义是不带 extern 的，而在其他使用这个变量的文件中，都需要有带 extern 的说明，表示此变量已在别处定义过了。例如，有一个由两个文件组成的程序如下：

file1.c：

```
int x ; /* 定义外部变量 x */
main()
{
    extern int y ; /* 说明 y 为外部变量 */
    ⋮
}
func1()
{
    int y ; /* 定义自动变量 y */
    ⋮
```

```
}
file2.c:
int y ;/* 定义 y */
extern int x ; /* 说明 x 为外部变量 */
func2 ()
{
    ⋮
    x ++ ; /* 操作外部变量 x */
    ⋮
}
func3 ()
{
    ⋮
    printf("%d", x) ; /* 操作外部变量 x */
    ⋮
}
```

在文件 file1.c 中定义了外部变量 x,在 file2.c 中每个函数都要使用到它,因此在文件开始部分说明了

```
extern int x ;
```

在文件 file2.c 中定义了外部变量 y,file1.c 中 main 函数使用这个外部变量,故在 main 中说明

```
extern int y ;
```

而函数 func1 中的

```
int y ;
```

不带 extern 说明,所以它是一个自动变量,作用域为 func1 内。

在大型程序中,为了使多个外部变量的说明一致,避免重复或遗漏,可以把它们合到一起放在一个文件中,在使用它们的文件开始处写上文件包含行

```
#include "文件名"
```

此文件在编译预处理时会自动地插入到相应的源文件中。这在协调分别编译的文件时非常有效。

最后说一点,外部变量提供了一种在函数间自由传递数据的机制,为编制程序带来了一些方便,但它也有很大的副作用,它破坏了函数的封闭性,使程序的控制复杂起来,所以,除非必要,不应过多使用外部变量

6.5.4 静态变量

静态变量分为内部(自动)静态变量和外部静态变量,在函数中定义的是内部静态变量,在函数外定义的是外部静态变量,内部静态变量的作用域是它所处的函数(或分程序),外部静态变量的作用域是它所在的文件。在变量名及其类型之前加上关键字 static,就规定该变量的存储类为静态的。

一个内部静态变量的存储形式是这样的:当第一次调用该变量所在的函数时,系统为它

分配存储单元，当控制从函数退出时，并不释放静态变量所占的存储单元，其值也仍然保留，下次再调用时，静态变量仍拥有上次调用时留下的值。

下面的程序可能会帮助你理解静变量的概念，做为比较，我们同样操作自动变量。

例 6.25

```
/*一个关于静态变量和自动变量比较的程序*/
main()
{
    void auto_static();
    int i;
    for (i = 0; i < 5; i++)
        auto_static();
}
void auto_static()
{
    int auto_var = 0;
    static int static_var = 0;
    printf ("auto_var=%d,static_var=%d\n", auto_var, static_var);
    auto_var ++;
    static_var ++;
}
```

运行结果

```
auto_var=0,static_var=0
auto_var=0,static_var=1
auto_var=0,static_var=2
auto_var=0,static_var=3
auto_var=0,static_var=4
```

main 函数共 5 次调用了 auto_static，每次调用自动变量 auto_var 都输出 0 值，这是因为每次调用时系统都给 arto_var 分配存储单元并赋初值 0，虽然

```
auto_var ++;
```

使 auto_var 增加到 1，但 auto_static 运行结束时，分配给 auto_var 的存储单元就被释放，下次还是重新开始，而 static_var 只是在第一次被调用时才分配给存储单元并赋初值 0，语句

```
static_var ++;
```

使 static_var 增加了 1，当 auto_static 结束运行时，static_var 的值仍然保留，再次调用 auto_static 时，系统不再为它重新分配存储空间并赋初值，而是采用以前留下的值，因此 static_var 的值每次调用增加 1。

有关静态变量的应用，一个很好的例子是求伪随机数。我们知道，掷骰子可以得到一个 1～6 之间的数，但具体是多少事先是不知道的，这就是随机数。计算机中可以利用一些算法求随机数，有些系统中提供了这类函数。我们也可以自己编一个产生随机数的函数，最常用的方法是线性同余法。这时一个随机数可以用前一个随机数求得：

$r_i = (multiplier * r_{i-1} + increment) \% modulus$

实际上这不是一个真正的随机数，因为我们可以通过 r_{i-1} 预知 r_i，但如果几个参数选得好的话，从效果上看还是很象随机数的，因此可以把这样产生的数叫做伪随机数。我们可以这样选择各参数。

modulus=2^{16}=65536

multiplier=25177

increment=13849

并选择一个最早的值作为种子，由它产生下面的伪随机数。这里选

seed=17

程序如下：

例 6.26 求伪随机数

```
/* 伪随机数发生器 */
#define MODULUS 65536
#define MULTIPLIER 25173
#define INCREMENT 13849
#define INIT_SEED 17
int random()
{
    static int seed = INIT_SEED ;
    seed = (MULTIPLIER * seed + INCREMENT) % MODULUS ;
    return(seed) ;
}
```

这里只是一个函数，如果你要试一试它，可以编一个 main 函数调用它。

上面是有关内部静态变量的讨论，下面我们来看一看外部静态变量，外部静态变量具有这样的性质：它的作用域从定义处起直到所在文件的尾部，对其他文件及本文件定义之前的部分都是不可见的，即使你加上 extern。也就是说它有很不错的隐蔽性。使用外部静态变量可以起到几个函数共享数据，而对其他函数保密的作用。下面我们把伪随机数发生器扩展成两个函数，它们都使用同一个 seed。

例 6.27

```
#define MODULUS 65536
#define MULTIPLIER 25173
#define INCREMENT 13849
static int seed ;
void init_seed(s)
int s ;
{
    seed = s ;
}
/* 返回一个 0～MODULUS－1 之间的整数 */
```

```
int random()
{
    seed = (MULTIPLIER * seed + INCREMENT) % MODULUS;
    return(seed) ;
}
/*返回一个 0～1 之间的实型伪随机数*/
double probability()
{
    seed = (MULTIPLIER * seed + INCREMENT) % MODULUS ;
    return(1.0 * seed / MODULUS) ;
}
```

这段程序使用的时候最好放在一个文件的尾部，这样可以把实现的细节隐藏起来，调用时只使用它们提供的伪随机数，另外在第一次使用之前应为 init_seed 提供一个原始的种子 s。

静态存储器也适用于函数。一个带有 static 的函数。作用域只是本文件，也就是不能从其他文件中调用静态存储函数。利用静态存储类，我们可以把与其他文件无关的变量或函数隐藏在文件内，使一个文件成为功能独立的模块，外部只能通过一些专门留出来的接口使用它。

6.5.5 寄存器变量

计算机中只有寄存器中的数据才能够直接参加运算，而一般变量是放在内存中的，变量参加运算时，需要先把变量的值从内存中取到寄存器中，然后计算，再把计算结果回放到内存中去。为了减少内存访问，提高运算速度，C 语言允许定义所谓寄存器变量，即希望用寄存器来做变量的存储单元，这可用关键字 register 来说明，如：

```
register i ;
```

寄存器变量只能在函数中定义，并只能是 int 或 char 型。一般只有使用最频繁的变量才定义成寄存器变量，如循环控制变量等。所以寄存器变量经常以下面的形式出现。

```
{
    register int i ;
    for (i = 0; i < n; i ++) {
        ⋮
    }
}
```

需要说明的是，计算机中可供寄存器变量使用的寄存器数量很少，有些机器甚至根本不许变量在寄存器中存储，当系统没有足够的寄存器时，register 类的变量就当做 auto 类来看待。

第六节　常见错误

1. 在函数定义后加分号。

如

```
int f(a, b) ;
int a, b ;
{
    ⋮
}
```

这在编译时，系统将指出错误。函数定义的括号后面不能用分号，因为这不是一个函数调用。由于语句后面要加分号，一不注意就把所有的行尾都加上了分号。

2. 非整型函数前没加类型标识符。

由于整型函数的类型标识符可以省略不写，而整型(int 和 char)函数在 C 中使用的又非常频繁，就容易渐渐地忘记非整型函数是要加类型标识符的。如我们自己写一个求平方根的函数 squ_rt 不可写成

```
squ_rt(x)
float x ;
{
    ⋮
}
```

由于省略类型表示整型，则返回时总是给一个整数值，这样 3 的平方根竟然会得 1，而且程序不会有任何有关的语法错误。一个好的习惯是：即使是 int 型也总是明确地写出，这样你就会习惯地给每个函数以必要的类型。

3. 形参说明写在函数体内。

如

```
int max(a,b)
{
    int a, b ;
    ⋮
}
```

这在编译时会产生错误，a,b 是形参，要写在花括号的外面，不要一写变量说明就总习惯写在花括号的后面，正确的写法是

```
int max(a, b)
int a, b ;
{
    ⋮
}
```

4. 调用还未定义的非整型函数而未加说明，如

```
main()
{
    float a, b, c ;
    a = 1.5 ;
```

```
    b = 2.3 ;
    c = fadd(a, b) ;
        ⋮
}
float fadd(x, y)
float x, y ;
{
    ⋮
}
```

编译时系统会指出错误，fadd 是非整型函数，如调用在先，定义在后，则应在调用之前说明它的类型，如可以在 main 之前，或 main 中说明部分加上

```
float fadd() ;
```

或者索性把 fadd 写在 main 之前。

5. 忽略参数的求值顺序。

如有语句

```
printf("%d", sub(i, --i) ) ;
```

而 sub 为

```
int sub(x, y)
int x, y ;
{
    return(x - y) ;
}
```

希望结果为 1，但实际中可能结果为－1，这是因为函数求值顺序可能是从右到左的，解决这种问题的办法是避免变量和它自身增减 1 的表达式同时做为一个函数的参数，实际上你只要将 printf 写成

```
printf("%d", sub(i, i - 1) ) ;
```

就不管是从右向左，还是从左到右计算参数值，都能得到希望的结果。

第七章　指　　针

本章学习重点

本章讨论C语言的指针类型，介绍指针变量、指针运算、指针与函数参数、指针与数组的关系，以及指针数组和指向指针的指针等概念。学习本章应熟练掌握指针的概念，掌握有关指针的各种操作，并能灵活地运用。

有人说C语言灵活方便，功能很强，也有人说C语言不容易学，容易出错，而专家对C语言的评价是：入门容易得道难。实际上，这些话多数情况下都是在谈论指针。指针是C语言中一个非常重要的概念，正确而灵活地使用指针，能够有效地描述各种复杂的数据结构，能够动态地分配内存空间，能够方便地操作字符串，能够自由地在函数间传递各种类型的数据，能够使程序更简洁紧凑，执行效率更高。许多程序员之所以喜欢C，有很大程度是因为指针。指针可以说是C语言中最有特色的内容，也是最具光彩的地方。因此，每一个想掌握C语言的人都应该深入研究指针，灵活准确地运用它，真正掌握C的精髓，从入门到得道。

第一节　指针变量

7.1.1　指针的概念

为了解指针的概念，让我们先来看一看数据在内存中是如何存储、如何读取的。

当我们定义了一个变量时，编译程序就会为它分配一定的存储单元。存储单元的大小由变量类型决定。例如在PC机上，一个int型数据占据两个字节，一个float型数据占据4个字节的存储空间。比如说，我们定义

```
int i, j, k ;
```

编译程序可能会为它们在内存中做如图7-1(a)形式的分配。

也就是说变量i占据以2000开始的两个字节，j占据从2002开始的两个字节，k占据从2004开始的两个字节。一但为变量分配了存储单元，我们以后对变量的操作就是针对内存单元的了，如

```
i = 10 ;
j = 15 ;
```

将整数值10送入2000开始的地址单元，将整数值15送入2002开始的地址单元。而

```
k = i + j ;
```

则是将2000中存放的值和2002中存放值取出相加，然后放到2004开始的单元中去。这个赋值语句执行完后的情况如图7-1(b)所示。这种按变量地址存取变量值的方法称为“直接访问”方式。

我们也可以用另一种方式访问变量，如图7-2。

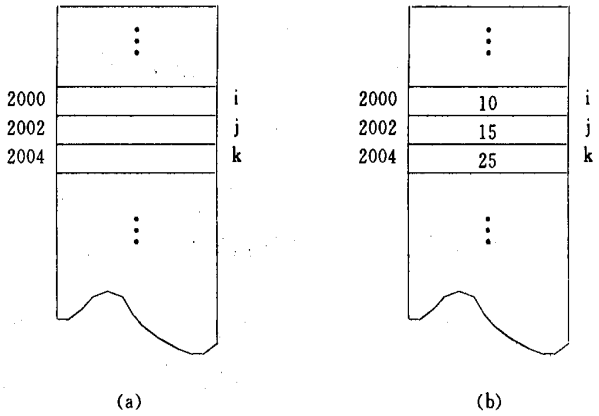


图7-1 直接访问

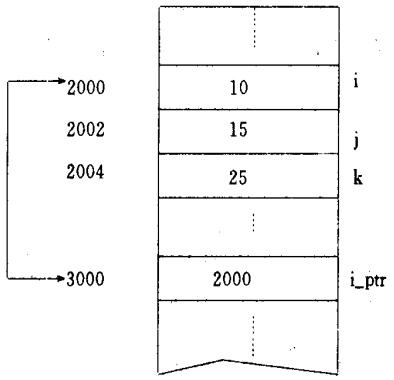


图7-2 间接访问

要访问变量i时，我们不是直接访问2000单元，而是通过3000单元间接地访问它。这时3000单元中放的是2000，也就是变量i的地址值。访问i时，先找到3000单元，从中取出变量存放的地址2000，然后通过这个地址找到变量i的值，也就是存在2000单元中的值10。这种方式称“间接访问”。内存单元3000在这里起什么作用呢？它存放的是指向另一个变量的地址，这实际上相当于它指向另一个变量。这种存放指向其它变量的地址的变量叫做指针变量，其中的地址值就是指针。所谓指针实际上就是地址，在多数情况下，该地址是内存中另一个变量(或函数等)的存储位置。

7.1.2 指针变量的定义

在C语言中允许使用指针类型的数据。指针变量的一般定义形式为

类型标识符 * 变量名

如

```
int *i_ptr ;
```

定义了一个指向整型值的指针变量i_ptr，我们也可以定义指向其他类型的指针，如

```
char *s1, *s2 ;
float *fptr ;
```

7.1.3 指针运算符

C语言为指针专门设置了两种运算符。

1. & 运算符

这是取地址运算符，它是单目运算符，它返回其后所跟操作数的地址，如

```
int i, *i_ptr ;
i_ptr = &i ;
```

将变量 i 的地址(注意，不是 i 的值)赋值 i_ptr。这个赋值语句可以理解为 i_ptr 接收 i 的地址。如果给 i 分配的地址是 2000 开始的单元，则赋值后 i_ptr 的值是 2000。

2. * 运算符

这是间接运算符，它也是单目运算符。* 把它的操作数当地址来对待，并访问那个地址以操作所需要的值。如果变量 i 的值是 10，i_ptr 中存放 i 的地址的话，赋值

```
j = *i_ptr ;
```

将把 i_ptr 中存放的值当做地址，然后取那个地址(i 的地址)中的值赋给变量 j，结果是把 10 赋给 j。

指针操作对初次接触的人来讲可能不太好理解，下面通过几段小程序来说明一下。

(1)

```
int i, *p ;
i = 100 ;
p = &i ;
```

如果给 i 分配的地址是 2000，给 p 分配的地址是 1500 的话，则定义之后的情况如图 7-3(a)，赋值 i = 100 ；执行后，情况如图 7-3(b)，这时 p 仍没有指向任何有效的位置，赋值 p = &i ；取变量 i 的地址值送给 p，使 p 指向变量 i，情况如图 7-3(c)。其实，一般人们并不关心具体的地址值，只要知道指针指向哪个变量即可。于是我们把图 7-3(c)改成 7-3(d)。

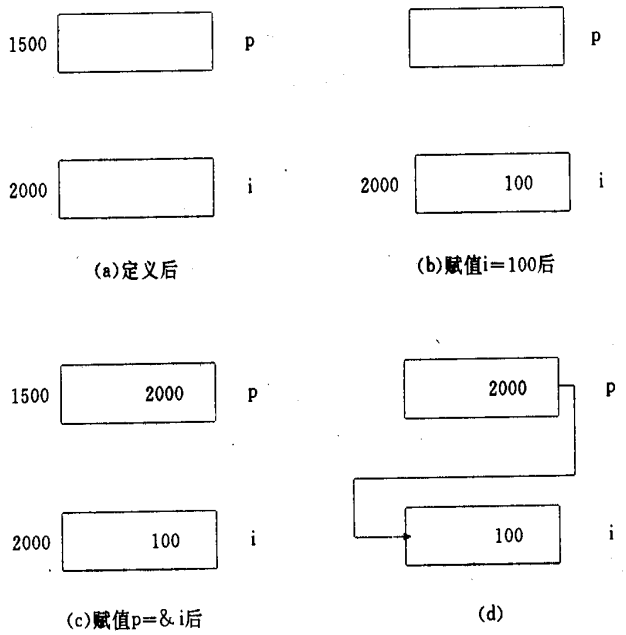


图7-3

(2)

```
int i, j, *p ;
p = &i ;
*p = 100 ;
j = *p ;
```

这里定义了整型变量i、j和指向整型的指针p，第一个赋值p = &i ；使p指向变量i，结果如图7-4(a)所示。第二个赋值*p=100；将100送给p所指的那个变量，也就是送给了i，如图7-4(b)，这时*p相当于一个整型量，所以赋值号两边是同类型。实际上

```
int *p ;
```

这种形式正说明*p的整体可以看成是一个整型变量。一但p指向哪个变量，对*p的操作就相当于对那个变量的操作。现在它指向i，所以*p这时就相当于i，于是把100送到了i中。同样，赋值

```
j = *p ;
```

把p所指的变量中的内容送给j，如图7-4(c)。

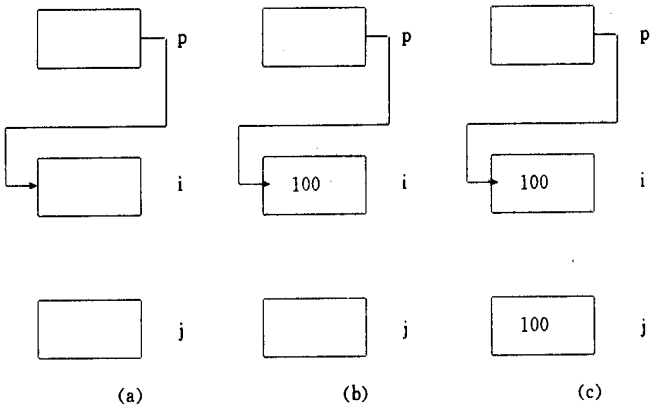


图7-4

要注意的是，指针在使用之前必须先使它指向某个具体的地方，如没有p = &i ；则p的指向是不确定的，虽然你仍可以写*p=100；但100却不知存放在什么地方。图7-5说明了这种情况。

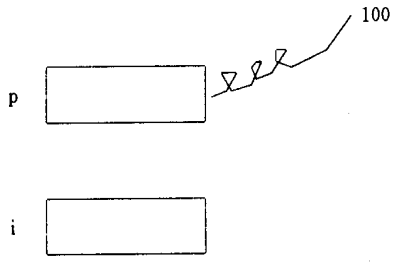


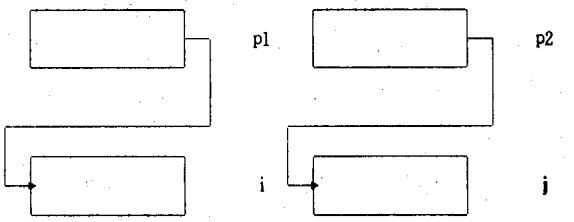
图7-5

(3)

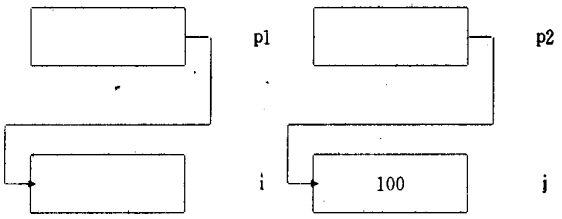
```
int i, j, *p1, *p2 ;
p1 = &i ;
```

```
p2 = &j ;
i = 100 ;
*p2 = *p1 ;
```

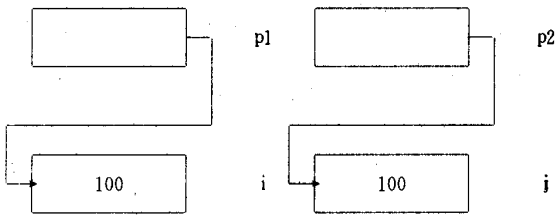
执行的情况如图 7-6。



(a)p1=&i,p2=&j



(b)i=100



(c)*p2=*p1

图7-6

最后一个赋值 *p2 = *p1 ;将 p1 所指的变量中的内容送到 p2 所指的变量中去。这个赋值这个时就相当于

```
j = i ;
```

(4)

```
int i, p1, p2 ;
p1 = &i ;
p2 = p1 ;
```

这里先将指针 p1 指向 i,如图 7-7(a),然后执行 p2 = p1 ;将 p1 的值也就是 i 的地址赋给 p2,这实际上使指针 p2 也指向变量 i,如图 7-7(b)所示。

由此可以看到,向一个指针变量赋值,就相当于让指针指向某一个变量的地址。

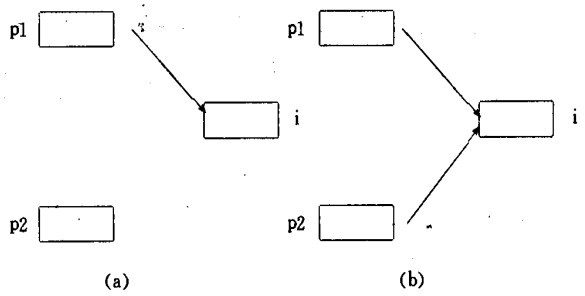


图7-7

第二节　指针与函数参数

在上一章中我们已经介绍了函数的概念，现在来试着编一个函数，以解决前面多次用到过的两个数值互换的问题。你的第一个直觉可能会使你写出如下的程序。

例 7.1

```
main() /* 测试 swap 的程序 */
{
    void swap() ;
    int a = 10, b = 20 ;
    printf("%d,%d\n", a, b) ;
    swap(a, b) ;
    printf("%d,%d\n", a, b) ;
}
void swap(x, y) /* x 与 y 互换 */
int x, y ;
{
    int temp ;
    temp = x ;
    x = y ;
    y = temp ;
}
```

运行一下

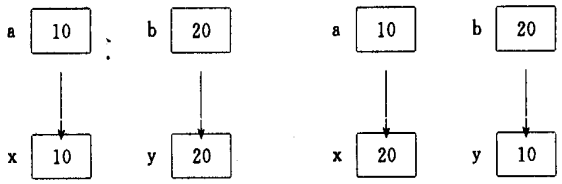
```
10,20
10,20
```

怎么回事？它们并没有交换。回忆一下，你可能想到：C 语言的函数参数都是传值的。现在让我们一起来看看程序的执行过程。主函数 main 调用函数 swap 时，将参数 a、b 的值传送给 swap 的形参 x、y，这相当于有赋值

```
x = a ;
y = b ;
```

情况如图 7-8(a)，x、y 接收到数值后，在 swap 中互换，由于参数是传值的，被调函数中形参

的改变并不影响到对应的实参，所以互换后的结果如图 7-8(b)，实参 a、b 的值并没有任何改动。这样，当 swap 运行完返回 main 后，printf 输出的当然还是原来 a、b 的值。



(a)参数传递　　(b)形参互换之后

图7-8

那怎么办呢？把 swap 定义为整型，用返回值将结果带回？这显然不行。函数只能返回一个值。你可能想到可以用外部变量。是的，外部变量可以解决这个问题，但这样简单的问题都要使用外部变量，那么更多的函数回送多个值呢？大量使用外部变量降低了程序的可读性，增加了程序出错的可能。看来这也不是好办法。那怎么办？可以利用指针！我们把例 7.1 改写一下

例 7.2

```
main()
{
    void swap();
    int a=10, b=20 ;
    printf("%d,%d\n", a, b) ;
    swap(&a, &b) ;
    printf("%d,%d\n", a, b) ;
}
void swap(x,y) /*x、y 互换*/
int *x, *y ;
{
    int temp ;
    temp = *x ;
    *x = *y ;
    *y = temp ;
}
```

执行一下这个程序

```
10,20
20,10
```

这回对了。现在让我们来看一看例 7.2 是怎么执行的。main 调用 swap 时，参数是变量 a 和 b 的地址，现假定 a 的地址是 1500，b 的地址是 2000，如图 7-9(a)。参数传递时仍然是传值的，但这回传递的是地址值，也就是把地址值 1500 和 2000 传给相对应的形参。swap 的形参这回定义为指向整型的指针，它们正好可以接收整型变量 a、b 的地址值，所以参数传递相当于有语句

```
x = &a ;
```

```
y = &b ;
```

参数传送后的情况如图 7-9(b)。也就是说指针变量 x、y 分别指向变量 a 和 b。当变量 temp 定义之后，语句

```
temp = *x ;
```

将 x 的值 1500 做为地址，取 1500 号单元中的存放的值赋给 temp，也就是将 10 赋值了 temp，如图 7-9(c)所示。语句

```
*x = *y ;
```

将 y 所指的变量(地址 2000 单元)的值赋给 x 所指的变量(地址 1500 单元)，这时地址 1500 单元中存有 20，如图 7-9(d)所示。最后一个赋值

```
*y = temp ;
```

将 temp 中暂时存放的值赋给 y 所指的变量(地址 2000 单元)，这样 2000 单元中存有数据 10，如图 7-9(e)所示。swap 执行完后，它的形参和内部变量被释放，但地址 1500 和 2000 中仍存有数据，它们就是变量 a 和 b 的值，如图 7-9(f)。这时从 main 中再次输出 a、b，其值是已经交换过的值了。

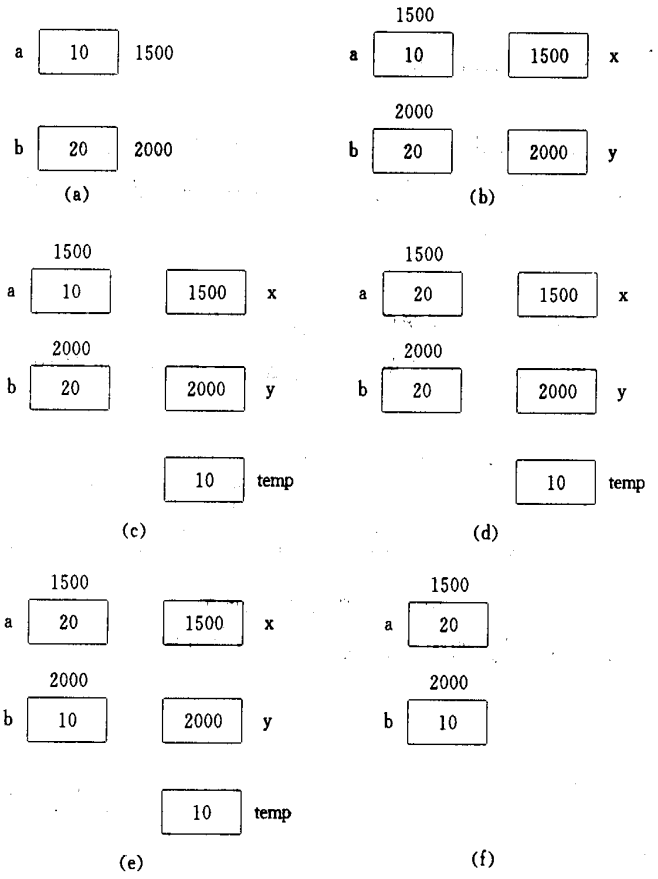


图7-9

从这个例子中我们可以看到：虽然 C 语言的函数参数都是传值的，但是可以通过地址值间接地把被调函数的某些数值传送给主调函数。这样指针又为我们在函数之间传递数据

提供了一种新的途径。需要理解的是，即使是指针作为参数，参数的传递仍然是传值的，形参改变的只是它所指的变量的值，而不是形参自身的地址值。正是因为地址值没有改变，我们才间接地将改变参数造成的影响传递到主调函数。

实际上，利用指针传递地址以获得返回值的情形早已用过，这就是 scanf 函数。我们在用 scanf 接收数据时，给出的变量前经常都带有一个取地址运算符 &，如

```
int i ;
char ch ;
float a ;
scanf("%d%c%f", &i, &ch, &a);
```

这时的变量 i、ch、a 前面都带有取地址运算符。当从输入流中获取相应的数值时，就将值放到这些变量所在的地址中，函数调用完后，程序中就得到了所需的值。

需要返回单个变量的值时可以用指针传送变量的地址。如果是一个数组中的元素需要在被调函数中改变，又应该怎么样呢？在前一章中讨论数组做为函数参数时，我们曾经介绍过一个数组名，实际上表示那个数组的首地址，就是说使用数组名作为参数就可以了。下面我们再做更细致的讨论。

对于单个变量，不管你是否连续定义，如

```
int i, j, k ;
```

系统也没有义务一定为它们安排连续的存储单元。如 i 可能分配 2000 单元，j 可能分配 2500 单元，而 k 可能分配 1500 单元，也就是说它们是相互独立的。对于数组则不同，系统必须为数组的所有元素安排连续的存储空间，如

```
int a[10] ;
```

系统可能为它安排从 5000 开始的存储单元，这样 a[0]占据 5000 和 5001(假定 int 型占 2 个字节)，那么 a[1]就占据 5002 和 5003 单元，a[9]必然占据 5018 和 5019 单元，如图 7-10 所示。

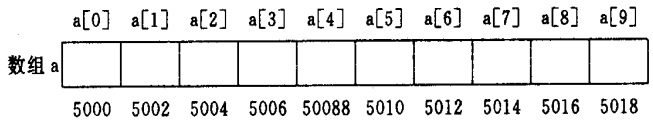


图 7-10　数组的存储形式

对于多维数组情况也是类似的，如图 7-11 给出了一个二维数组

```
int b[5][5] ;
```

的存储情况。

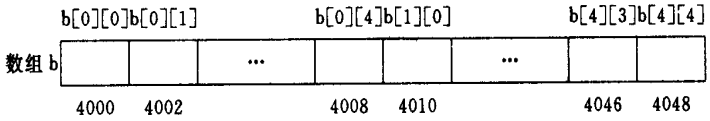


图 7-11　多维数组的存储形式

也就是说，一个数组只要确定了它的起始地址，其余各元素的地址就都是固定的，也就可以对数组中所有元素进行操作了。

由于这个原因，数组做为函数参数时，我们只需要给出做为数组起始地址的数组名就可

以了，这时被调函数中对数组的任何改变，都将影响到主调函数。因为被调函数和主调函数中的数组都是在相同的存储单元中操作的。如主调函数中有

```
int a[10] ;
func(a) ;
```

被调函数为

```
void func(b)
int b[10] ;
{
    ⋮
}
```

则函数 func 被调用时，内存的情况如图 7-12。

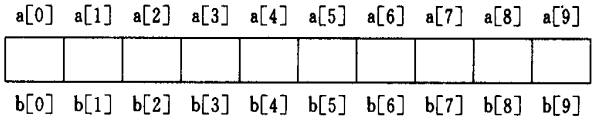


图 7-12　数组做为参数的情形

这样数组 b 中元素的改变自然就要影响到数组 a，因为它们共用同一段存储空间。

例 7.3 快速排序

快速排序是目前使用的较好的排序算法。它是由 C. A. Hoare 发明并命名的。它是这样操作的：先在要排序的数中选出一个作为比较标准的数，然后把余下所有的数都与它做大小比较，凡比它大的数都放在它的一侧，而比它小的数都放在另一侧，经过一遍比较后，实际上已确定了这个数的最终位置。然后对它两侧的数据再分别进行上面的操作。例如有数据

5　8　1　9　6　3　4　0　7　2

首先选择中间的数 6 做为比较数，然后逐个比较其他数：5＜6，不交换，8＜6，记住 8 的位置，再从后面向前比较，2＜6，次序不对，这时 8 与 2 互换，得到

5　2　1　9　6　3　4　0　7　8

再从刚换完的下一个位置继续比较，1＜6 不换，9＞6，记住位置，从后面比较，7＞6，不换，0＜6，做 9 与 0 的互换，得到

5　2　1　0　6　3　4　9　7　8

由于 6 前面的数都已小于 6，所以只对其后的数做比较，从 4 开始，由于 4＜6，则 6 与 4 互换，得到

5　2　1　0　4　3　6　9　7　8

还剩一个 4 后的数没有比较，再与 6 比较，3＜6 并且 3 在 6 的左侧，不必互换，于是一遍比较结束。这时，6 前的数都小于 6，而 6 后的数都大于 6，这就是一遍扫描的结果。6 已经确定了它最终的位置，下面可以对它两侧的数据分别重复上面的操作，最后就可以使全部数据按升序排列。从以上的分析可以看出，快速排序最好是用递归的方法求解，其参数应该有存放数据的数组和排序的起始及终止位置（并不总是从 0～n－1），因此一共需要三个参数，如

```
void quick (v, left, right)
```

在主调函数中可用

```
quick(d, 0, n-1) ;
```

形式调用。但应该考虑,许多排序方法只需要两个参数,如 shell 排序形如

```
void shell(v, n)
```

为了统一,我们最好也采用接口只有两个参数的形式,如

```
void quick(v, n)
```

这时可在 quick 中再调用

```
qs(v, 0, n - 1)
```

来实现快速排序。

程序如下:

```
/* 快速排序的接口 */
void quick(v, n)
int v[],n;
{
    void qs() ;
    qs(v, 0, n - 1) ;
}
/* 快速排序 */
void qs(v, left, right)
int v[],left,right ;
{
    int i, j, x, temp ;
    i = left ; /* 左侧位置 */
    v = v[(left + right) / 2] ; /* 取一个比较数 */
    while (i < j) {
        while (v[i] < x && i < right)
            i ++ ; /* 跳过左侧比 x 小的值 */
        while (v[j] >x && j < left)
            j -- ; /* 跳过右侧比 x 大的值 */
        if (i <= j) {
            temp = v[i] ; /* 互换 */
            v[i] = v[j] ;
            v[j] = temp ;
            i ++ ; /* 移动位置 */
            j -- ;
        }
    }
    if (left < j)
        qs(v, left, j) ; /* 左侧继续排序 */
    if (i < right)
```

```
        qs(v, i, right) ; /* 右侧继续排序 */
    }
```

在这个递归函数的例子中，数组 v 既做为形式参数，又做为实际参数。由于数组名代表地址这个性质，在每次调用时，实际上都是对同一段存储单元中的数据操作(只是数据的范围不同)，这就保证了数组中的数据能正确地排序。

第三节　指针与数组

7.3.1　指向数组元素的指针

C 语言中，指针与数组有着非常密切的关系，事实上指针大概最常来指向数组元素。例如

```
int a[10] ;
int *p ;
```

定义了一个整型数组 a 和一个指向整型量的指针 p，现在我们可以像把指针指向单个整型变量那样，让指针指向某个数组元素，如

```
p = &a[4] ;
```

将数组元素 a[4]的地址赋给 p，也就是让 p 指向 a[4]。如果希望让指针指向数组的第一个元素，可以写

```
p = &a[0] ;
```

在前面我们曾经讲过，一个数组名就是该数组的起始地址，也就是数组元素 a[0]的地址，因此

```
p = a ;
```

与

```
p = &a[0] ;
```

效果是一样的，都是将数组 a 的起始地址赋给 p(不是将数组 a 中各元素赋给 p)。所以有了上面的赋值后，*p 的内容就是 a[0]的内容。赋值 p = a ;的效果如图 7-13 所示。

7.3.2　通过指针引用数组元素

指针指向数组元素后，就可以通过指针来引用相应数组元素的值了。如

```
p = a ;
```

使 p 指向数组 a 的起始地址，我们可以用 *p 来引用 a[0]，如

```
*p = 1 ;
```

就是把整数值 1 赋给 a[0]。那么我们是否还可以通过指针 p 来引用数组 a 的其他元素呢?答案是肯定的。C 语言规定：一但指针指向某个数组元素之后，p+1 就指向同数组中下一个元素，而不管数组是什么类型，如图 7-14 所示。这样我们就可以用指针来访问数组中的任何元素了。如 *(p + 1)就是 a[1]，而一般 *(p + i)就是 a[i]。要注意，这里括号不可缺少，因为 * 的优先级比＋高，如不写括号，则 *p + 1 实际上是 a[0] + 1，显然含义是不一样的。

不要以为总是有 *(p+i)和 a[i]相等的对应关系，如果赋值是

```
p = &a[4] ;
```

则 p 指向 a[4]，p+1 指向 a[5]，而 p−1 指向 a[3]，也就是说，指针在数组中是可以移动的。

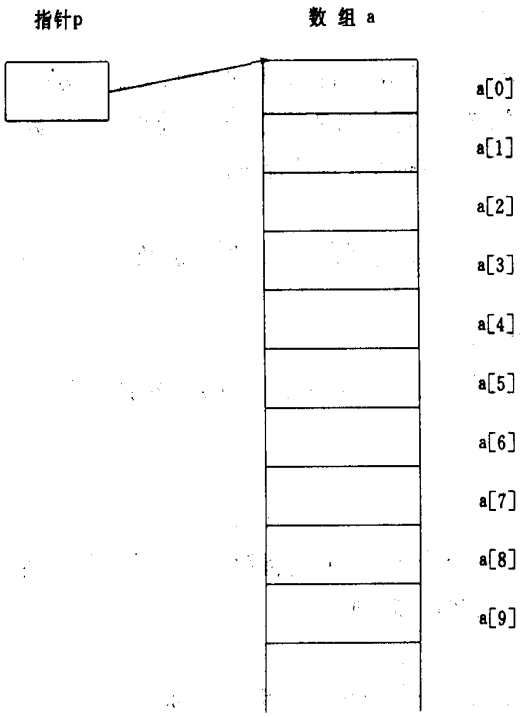


图7-13　指针数组元素的指针

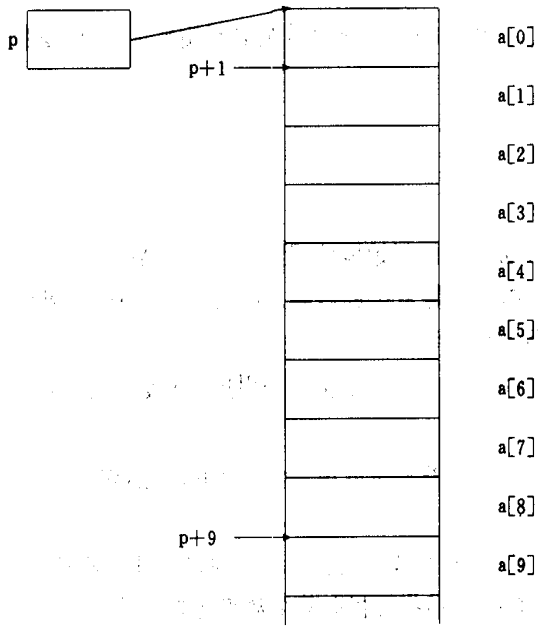


图7-14

通常在用指针操作数组元素时，我们可以通过指针变量的加1或减1运算来移动指针，如

```
p = a ;
```

使指针p指向a中的第一个元素a[0]之后，操作

```
p ++ ;
```

使指针 p 指向了 a[0]的下一个元素 a[1]，情况如图 7-15。

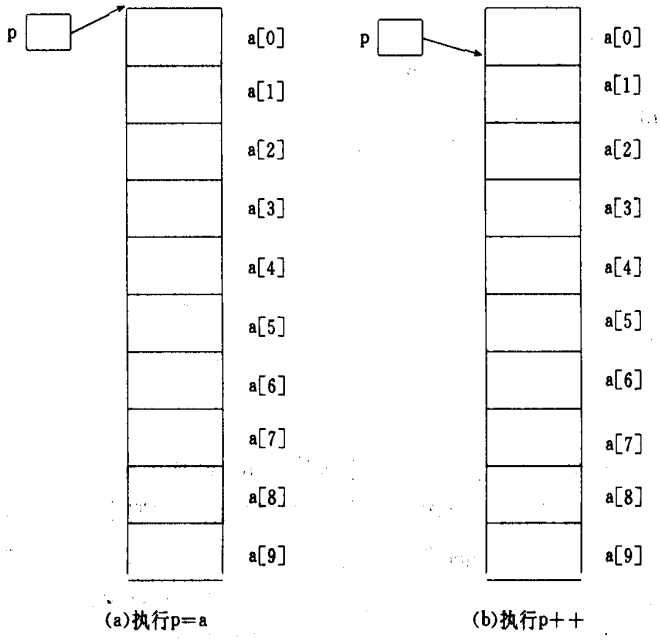


图7-15　指针的移动

当然你也可以对指针作减 1 操作，如

```
p --;
```

要注意的是，指针做加减运算时，应随时警惕，不要让指针指向数组以外。如指针指向 a[9]时再做 p++，将使指针指向数组 a 的范围以外。

另外，C 语言还允许以 a+i，*(a+i)这种形式操作数组，其中 a+i 相当于 &a[i]，也就是 a[i]的地址，自然 *a[i]就是 a[i]自身了。这种形式看起来与指针形式 p+i，*(p+i)相似，但你不能作 a++之类，使数组 a 的起始地址有所改变。

C 语言数组与指针关系密切，凡是能用数组下标操作的，都可以用指针来操作。下面的例子用两种办法解决同样的问题。

例 7.4 输出数组中的元素。

程序 1：数组下标版

```
main()
{
    int a[10], i ;
    for (i = 0; i < 10; i ++)
        scanf("%d", &a[i]) ;
    for (i = 0; i < 10; i ++)
        printf("%d", a[i]) ;
    printf("\n") ;
}
```

程序 2：指针版

```
main()
{
    int a[10], *p;
    for (p = a; p < (a + 10); p ++)
        scanf("%d", p);
    for (p = a; p < (a + 10); p ++)
        printf("%d", *p);
    printf("\n");
}
```

运行结果如下

```
9  8  7  6  5  4  3  2  1  0
9  8  7  6  5  4  3  2  1  0
```

程序1没有什么要多讲的，让我们看一下程序2：for循环中第一个表达式 p = a；使 p 指向数组 a 的第一个元素，以后 p 每次加1，使 p 指向数组 a 中的下一个元素，条件 p < (a + 10)中的(a + 10)应该是a[10]的地址，但数组 a 并没有 a[10]这个元素，所以实际上它是数组 a 以后的存储单元，也就是说在数组 a 的范围以外，这个条件保证了 p 在数组 a 的范围内移动。由于 p 是地址，所以在用 scanf 接收数据时不需要再加 &，如写成

```
scanf("%d", &p);
```

反而错了。类似理由，用 printf 输出数值时，需要用 * 操作符，而不能只写

```
printf("%d", p);
```

从这个例子中可以看到，用下标或用指针可以完成同样的工作，而且看起来用下标形式更容易理解。于是你可能会问：为什么要使用指针呢？它不是更麻烦吗？在这里确实更麻烦，但并不总是如此，这在后面一些例子中可以体会到。另一个很好的理由是，指针方法可以得到比下标方法更短的机器代码，而且执行也更快。不应忘记，C 语言是为编写系统程序而设计的，紧凑和快速是其追求的目标之一。

为了进一步了解指针的用法，我们把例 7.4 中指针版程序改写一下：

例 7.5

```
main()
{
    int a[10], *p, i;
    p = a;
    for (i = 0; i < 10 ;i ++) {
        scanf("%d", p);
        p ++;
    }
    for (i = 0; i < 10; i ++) {
        printf("%d", *p);
        p ++;
    }
```

```
    printf("\n") ;
}
```

这个程序使用非常明显的循环控制来代替不那么容易读的条件

```
p < (a + 10)
```

由于都是循环 10 次，所以你很容易想到这种办法，现在来运行一下

```
0  1  2  3  4  5  6  7  8  9
-20  285  1  -22  596  -18  0  14916  21596  23619
```

输出怎么会是这样？我们来分析一下。一开始

```
p = a ;
```

使 p 指向了数组 a 的起址地址。第一个 for 循环使 p 移动 10 次，也就是指向了 a[9]后面的地址，如图 7-16 所示。这时实际上 p 已经指向数组 a 以外的地方了。第二个 for 循环中再引用 p，处理的已经不是数组 a 中的内容了，很遗憾，C 语言没有指出错误(有时候指针超越数组范围会产生严重错误而造成死机)，而是继续操作，只是结果令人目瞪口呆。

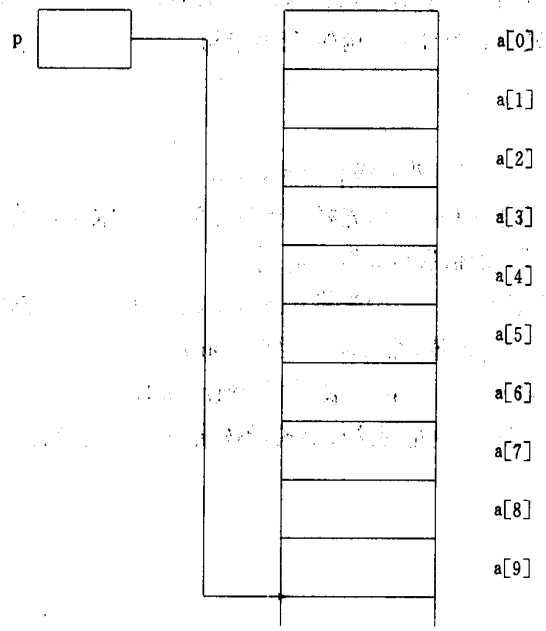


图7-16

从这里我们可以看到，当指针移动时，系统本身并不为你保证不越界(这不能不说是 C 的一个缺点)，这就需要自己多加小心了。看一下图 7-16，你会想到，在第二个循环之前，应该让 p 再次指向 a[0]，也就是需要加上语句

```
p = a ;
```

另外，C 语言倾向于写得简洁，我们也可以把 p 指针的移动写到 scanf 和 printf 中，于是，得到一个正确的程序。

```
main()
{
    int a[10], *p, i ;
    p = a ;
```

```
    for (i = 0; i < 10; i ++)
        scanf("%d", p ++) ;
    p = a ;
    for (i = 0; i < 10; i ++)
        printf("%d", *p ++) ;
    printf("\n") ;
}
```

运行结果

```
0  1  2  3  4  5  6  7  8  9
0  1  2  3  4  5  6  7  8  9
```

现在让我们来看看 *p++的含义，由于 * 与++有相同的优先级，而它们是右结合的，所以 *p ++等价于 *(p ++)，也就是先取 p，求 *p 的值，然后 p 加 1，这与

```
(*p) ++
```

是不同的。(*p) ++表示取 *p 的值，然后把这个值加 1，当 * 与++或--一起使用时，要特别小心，搞清楚不同形式的含义。例如，先有语句

```
p = &a[4] ;
```

则

(1) *p ++ 取 a[4]的值，然后将 p 指向 a[5]。

(2) * ++p 相当于 *(++p)先将 p 指向 a[5]，然后取 a[5]的值。

(3) *p -- 取 a[4]的值后将 p 指向 a[3]。

(4) * --p 相当于 *(--p)先将 p 指向 a[3]，然后取 a[3]的值。

(5)(*p)++ 取 a[4]的值，然后加 1，相当于 a[4]+1。

(6)-- *p 相当于--(*p) 取 a[4]的值后减 1。

如果你在开始时不能很好地把握这种形式，不妨把 * 和++(或--)操作分别考虑。如把

```
i = *p ++;
```

写成

```
i = *p ;
p ++ ;
```

通过指针和数组关系的讨论，我们知道，凡可用数组下标进行的操作都可以用指针完成。

指针甚至可代替数组来作为形参。如，

例 7.6 求数组中的最大值。

```
main()
{
    static int a[10] = {3,7,1,9,0,2,8,4,6,5} ;
    int max ;
    max = max_value(a, 10) ;
```

```
    printf("max=%d\n", max) ;
}
int max_value(x, n)
int *x, n ;
{
    int *p, m ;
    m = *x ;
    for (p = x + 1; p < x + n; p ++)
        if (*p > m)
            m = *p ;
    return(m) ;
}
```

运行结果

```
max=9
```

我们来分析一下函数 max_value，它的第一个参数是指向整型的指针，当 main 函数调用 max_value 时，将数组 a 的起始地址传给 x，这样 x 就指向了 a[0]，赋值

```
m = *x ;
```

把 a[0]的值赋给 m，p 指针从 x+1，也就是 a[1]处开始移动直到 x+n，也就是 a[10]的位置时结束。如果 *p>m，就是说后面的某个 a[i]大于以前的各值，则把它记录下来，最后返回 a[i]中最大的值. 图 7-17 表示这种操作的情况。

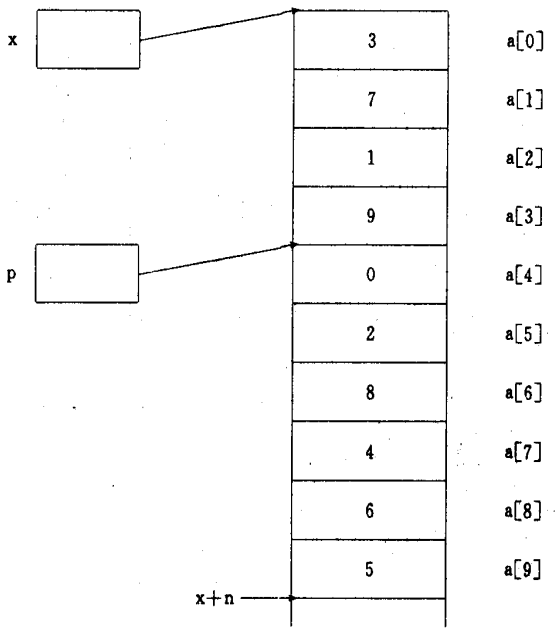


图7-17

我们也可以用指针的方式来调用函数，这时主函数 main 可以写成

```
main()
{
```

```
    static int a[] = {3,7,1,9,0,2,8,4,6,5} ;
    int max, *p ;
    p = a ;
    max = max_value(p,10) ;
    printf("max=%d\n", max) ;
}
```

程序运行的结果也是一样的。

利用指针与地址的关系，我们还可以只在部分数据范围内求最大值，如函数调用可以写成

```
max = max_value(p + 4, 6) ;
```

或用两条语句写成

```
p = &a[4] ;
max = max_value(p, 6) ;
```

其功能是从 a[4]开始求最大值，结果为 8。这时函数 max_value 中变量的情况如图 7-18。

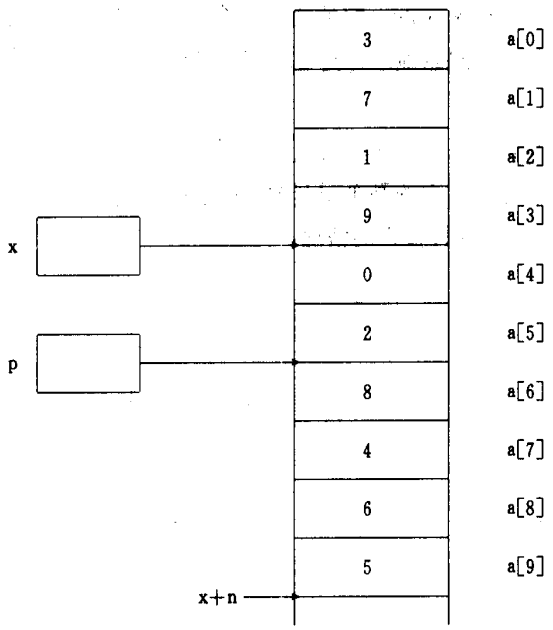


图7-18

当然你也可以不让求值范围到 a[9]，如

```
p = a ;
max = max_value(p + 4, 2) ;
```

将在 a[4]、a[5]中求最大值，结果是 2。

7.3.3 指针运算

指针指向数组后，允许有几种运算。

1. 关系运算

如有两个指针 p 和 q，它们指向同一数组，则它们可以进行关系运算，如

```
p < q
```

表示当 p 所指的元素排在 q 所指的元素之前时，关系表达式为真，

```
p ! = q
```

表示当 p 与 q 没有指向同一元素时为真。要注意，只有当 p、q 指向同一数组时，比较才是有意义的，假如 p、q 指向不同数组，则要失败。如果你的机器能够指出这种错误，那还真是幸运，如果它不产生任何错误提示，你根本就不知道程序怎么会产生一些莫名其妙的结果。所以作指针变量比较时，你必须清楚地知道它们是否已指向同一数组了。

2. 两个指针可做减法运算

当两个指针 p 和 q 指向同一数组时，它们可以相减，如

```
p - q
```

得到的值是一个整型量，表示 p 和 q 之间有多少个元素，同样 p、q 必须指向同一数组相减才有意义。

3. 指针可以与整数相加减

这在前面已经见过了，它表示数组中下几个或前几个元素，这时要记住的是，不要让指针超越数组的范围，而且指针也不能与浮点数相加减。

4. 可以用运算符 * 或 &

这一点是显而易见的。

除以上操作外，对指针进行其他操作都是非法。应该记得，指针就是地址，就象日常生活中的门牌号码似的，你可以比较两个门牌号码，也可以把两个号码相减看看中间还有几个号码，但把两个门牌号相加就没有任何实际意义，当然就更谈不上乘除之类了。

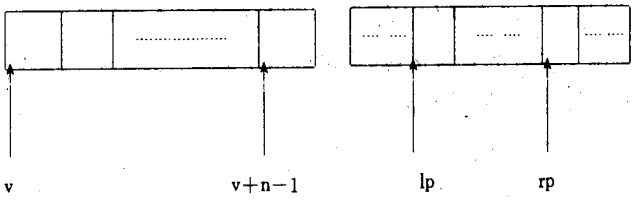
下面的例子给出了指针的几种运算，你可以从中思考一下。

例 7.7 用指针方式实现快速排序。

```
/* 快速排序接口，指针方案 */
void quick(v,n)
int *v, n ;
{
    void qs() ;
    qs(v, v + n - 1) ;
}
/* 快速排序，指针方案 */
void qs(lp, rp)
int *lp, *rp ;
{
    int *i, *j, *x ;
    int temp ;
    i = lp ;
    j = rp ;
    x = lp + (rp - lp) / 2 ; /* 指针运算 */
    while (i <= j) { /* 指针比较 */
        while ( *i < *x && i < rp)
```

```
            i ++ ;/ * 指针移动 * /
        while ( * x < * j && j > lp)
            j -- ; / * 指针移动 * /
        if (i <= j) { / * 指针比较 * /
            temp = * i ; / * 互换数值 * /
            * i = * j ;
            * j = temp ;
            i ++ ; / * 移动指针 * /
            j -- ;
        }
    }
    if (lp < j) / * 指针比较 * /
        qs(lp, j) ; / * 参数为指针 * /
    if (i < rp)
        qs(i, rp) ;
}
```

这里由于 qs 的参数使用指针，左右指针就限定了数组排序的范围，因此只需要两个参数就可以了。图 7-19 给出了 quick 中的调用情况和一般的调用情况。



(a)quick中调用形式　　(b)一般调用形式

图7-19

下面我们来看看程序中遇到的指针运算语句：

```
x = lp + (rp - lp) / 2 ;
```

用到了两种指针运算，它不能写成

```
x = (lp + rp) / 2 ;
```

这是因为 lp 和 rp 都是指针，两个指针相加得不到什么结果(编译时会出错)，这里用

```
rp - lp
```

求 rp 与 lp 之间有多少个元素，得到一个整型值，除以 2 后，得到 lp～rp 之间元素数目的一半，然后这个值与 lp 相加，就使指针 x 移到了排序数据的中间位置。程序中还多次用到了指针的比较和指针的移动，这在程序中用注释标出。由于是用指针而不是用下标，所以关于值的比较和值的互换都要使用运算符 * ，这是不能丢失的。

从前面的讨论可以看出，C 语言中的指针、数组以及地址的计算是一致的。事实上，把指针、数组和地址计算有机地结合起来正是 C 的主要优点之一。

第四节　字符指针

7.4.1　字符指针的定义和使用

指向字符类型数据的指针叫做字符指针，如

```
char *s;
```

定义了一个指向字符类型的指针变量 s，与前面讨论的一样，字符指针可以指向一个字符数组的元素。

例 7.8

```
main()
{
    static char c[] = "character string.\n";
    char *cp;
    cp = c;
    printf("%s", cp);
}
```

运行结果

```
character string.
```

程序中我们定义了一个静态字符数组并赋予初值，赋值

```
cp = c;
```

使字符指针指向字符数组 c 的第一个元素，如图 7-20 所示。输出时 printf 从 cp 所指的位置开始逐个字符输出，直到遇到字符串结束标志 '\0'。

如果你觉得用 printf 看不出字符指针是如何操作的，我们也可以采用一个一个字符输出的方法，现在把例 7.8 改一下。

例 7.9

```
#include "stdio.h"
main()
{
    static int c[] = "character string.\n";
    char *cp;
    cp = c;
    while (*cp != '\0')
        putchar(*cp++);
}
```

运行结果与例 7.8 一样。

这里指针 cp 在数组中移动，逐个输出字符，直到 while 条件为假。很显然，与逐个输出整型数组中的元素具有相类似的操作过程。

如果字符指针只能这样用，也就没有什么太稀奇的东西了。由于 C 语言设计的初衷是为了书写系统程序，而系统程序设计中经常要处理字符串，为了方便地处理字符串，C 语言

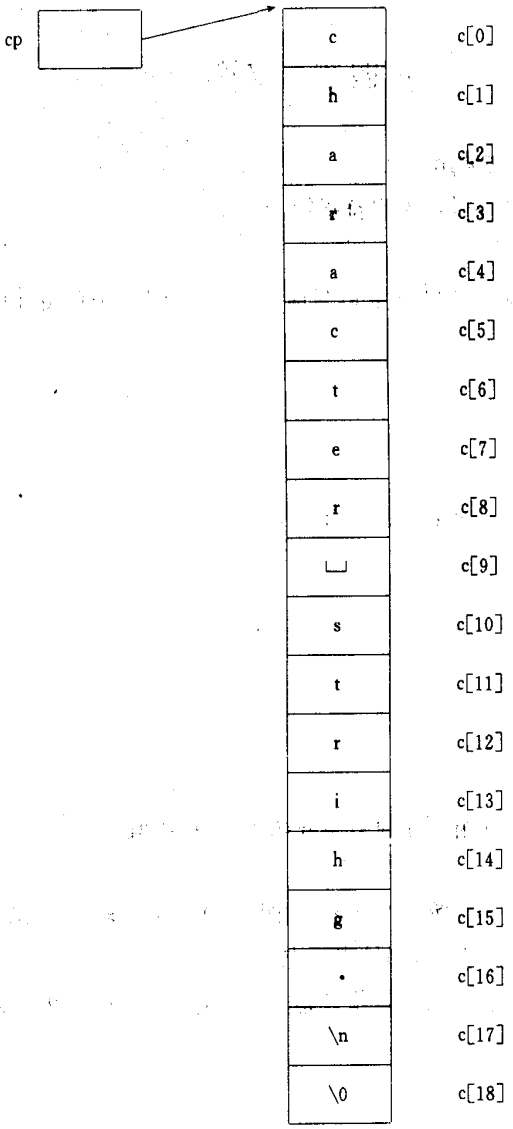


图7-20

的设计者赋给了字符指针更强的功能。

例 7.10

```
main()
{
    char *cp = "character string.\n";
    printf("%s", cp);
}
```

这里没有定义字符数组,而是直接把字符指针指向了一个字符串常数。实际上C语言对字符串常数的处理也是把它当做一个字符数组,为它开辟一片连续的存储空间,所不同的是这部分存储单元中的内容不能被改变。于是经定义和初始化后,内存中的情形如图 7-21 所示。

这时存放字符串的存储单元没有自己的数组名和下标,如果你还想用下标来操作,可以

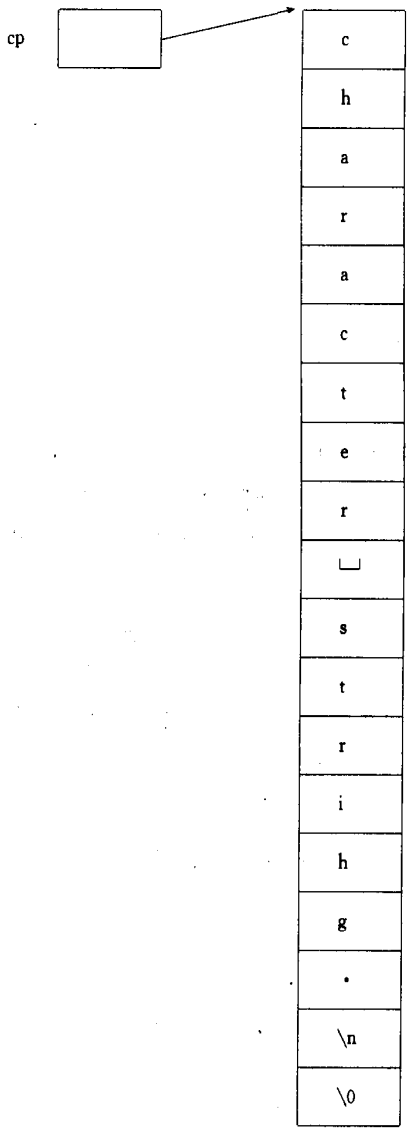


图7-21　指向字符串常数的指针

借用指针，这时相当于cp[0]存放着'c'；cp[1]存放'h'，…依此类推。虽然cp不是数组，但当它指向字符串时，你可以象操作一个数组那样去操作它。

定义

```
char *cp = "character string.\n";
```

也可以写成两行

```
char *cp;
cp = "character string.\n";
```

看到的效果就象是把一个字符串赋给一个变量。不过你不要真以为如此，赋值时只是把存放字符串常数的内存起始地址赋给了字符指针变量，这与以前对指针的赋值没什么两样。

定义了一个字符指针变量，并使它指向特定的位置后，我们也可以在scanf函数中使用它，就象使用字符数组似的。

例7.11

```
main()
{
    char  c[80], *str ;
    str = c ;
    scanf("%s", str) ;
    printf("%s\n", str) ;
}
```

运行结果

```
string
string
```

这里为什么还要一个字符数组呢？如果 str 没有显式地指定它的指向，则它所指的地方是无法确定的，可能根本就不是一个可访问的地址值。在用指针接收数据之前，必须先让它指向某个确定的内存区，然后才能使用。此外，由于 str 定义为指针，scanf 中自然也就不需要加 & 了。

由以上的讨论可以看出，虽然指针与数组有着密切的关系，但指针不是数组，数组在内存中占有一片连续的存储空间，可以存放许多数据，而指针变量则不同，它只是一个变量，只能存放一个值，而且只有当这个值为指定内存区的地址时，操作才有意义。

指针不是数组，但我们可以用指针来操作数组，这在字符指针和字符数组之间显得更方便。

例 7.12 将获取的输入行复制到输出。

```
#include "stdio.h"
#define MAXLEN 80
main()
{
    char line[MAXLEN + 1] ;
    while (getline(line, MAXLEN) >= 0)
        printf("%s\n", line) ;
}
int getline(line, maxlen) /* 获取一个输入行 */
char *line ;
int maxlen;
{
    char c ;
    int len ;
    len = 0 ;
    whlie ( (c = getchar() ) ! = EOF && c ! = '\n' && len < maxlen) {
        *line ++ = c ;
        len ++ ;
    }
```

```
    if (c == EOF)
        return(-1) ;
    *line = '\0' ;
    return(len) ;
}
```

这个程序中有意义的是函数 getline，main 只是为了测试一下 getline。函数 getline 利用指针构造出一个完整的行。main 中调用时，将数组的首地址传给 getline，getline 利用指针从数组的开始位置一个个地放入字符，这是由循环体中的语句 *line ++ = c ;完成的，当读入的字符不是 EOF(打 ^ z 的回送值)也不是换行符并且字符数组没超长时，就将这个字符送到指针 line 所指的位置，然后指针后移。当得到一个完整的行时，就加上字符串结束标志\0 并返回，这时在 main 中字符数组里就存有读入的输入行字串符了。

7.4.2 字符串指针用作函数参数

例 7.12 给出了指针做为形参的例子，同样指针也可以作为实参。我们可以用字符指针作为参数来传递字符串。下面这个例子就是用字符指针作为参数，在主调函数与被调函数之间传递字符串。

例 7.13

```
main()
{
    void str_cpy() ;
    char *s1, *s2 ;
    s1 = "Good morning" ;
    s2 = "How do you do" ;
    printf("%s\n%s\n", s1, s2) ;
    str_cpy(s2, s1) ;
    printf("%s\n%s\n", s1, s2) ;
}
/*拷贝 t 到 s*/
void str_cpy(s, t)
char *s, *t ;
{
    while ( (*s = *t) != '\0') {
        s ++ ;
        t ++ ;
    }
}
```

运行结果

```
Good morning
How do you do
Good morning
```

Good morning

在主调函数中，字符指针 s1 指向字符串"Good　morning"的起始地址，字符指针 s2 指向字符串"How　do　you　do"，情况如图 7-22(a)所示。函数 str_cpy 被调用时，s1、s2 作为参数传递给形参 t 和 s，这时的情形如图 7-22(b)所示。while 中的赋值将 t(也是 s1)所指的地址中的内容送到相应的 s(也是 s2)所指的地址中去，两个指针同时移动，当 * t 为'\0'时，赋值表达式

```
*s = *t
```

的值也是'\0'，这时循环的条件为假，循环结束。内存中的情形如图 7-22(c)所示。

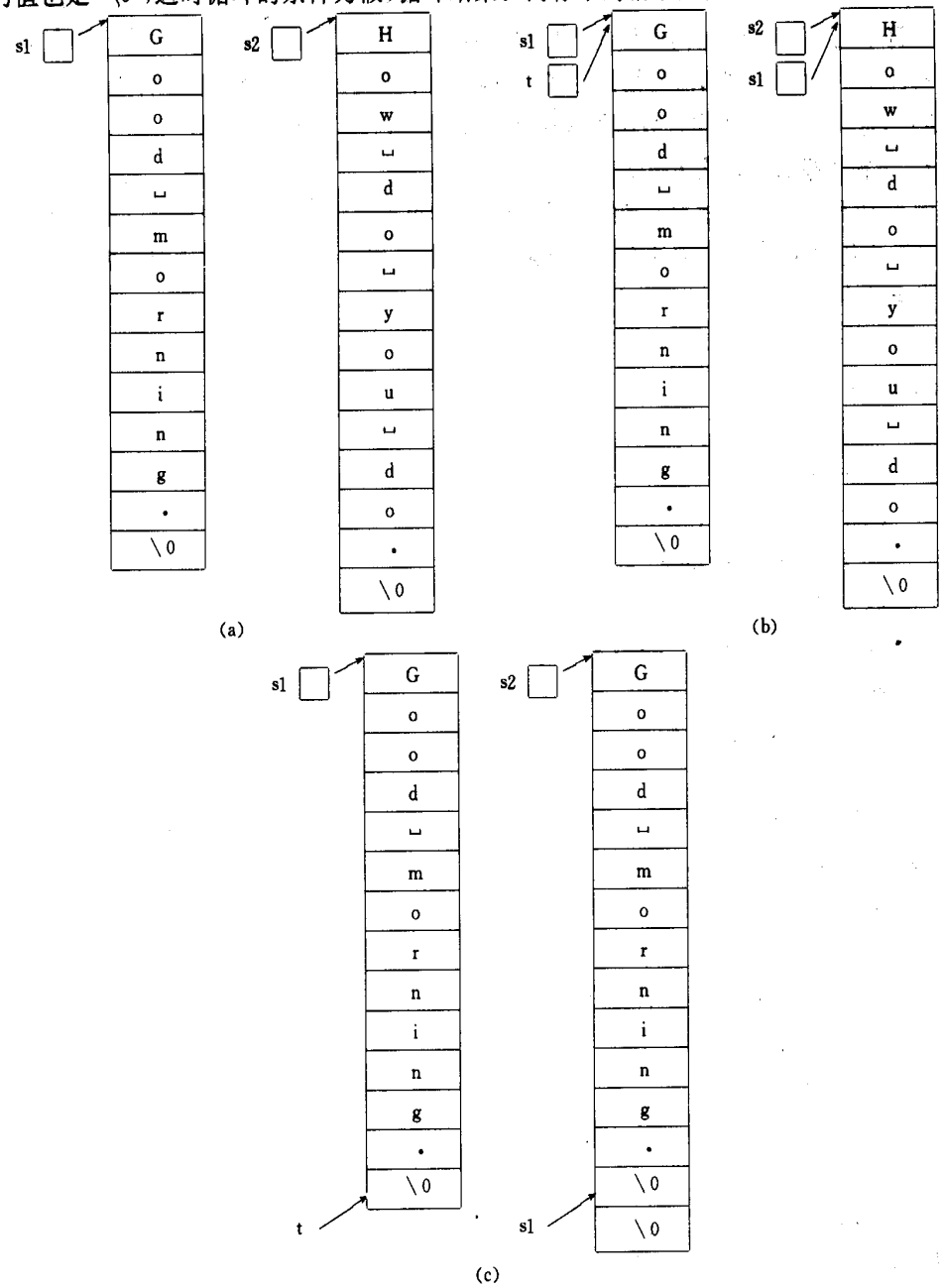


图7-22

你可能会觉得，对 s2 的赋值很牵强，既然我们要做的工作是把 s1 的内容复制到 s2 中去，有什么必要先对 s2 赋值呢？main 函数不能写成如下形式吗？

```
main()
{
    void str_cpy() ;
    char *s1, *s2 ;
    s1 = "Good morning" ;
    str_cpy(s2, s1) ;
    printf("%s\n%s\n", s1, s2) ;
}
```

答案是不行的。这里 s2 没有指向确定的存储空间，s1 的内容将不知道复制到什么地方，为此你需要将 s2 定义为字符数组，这样就可以写成

```
main()
{
    void str_cpy() ;
    char *s1, s2[20] ;
    s1 = "Good morning" ;
    str_cpy(s2, s1) ;
    printf("%s\n%s\n", s1, s2) ;
}
```

C 语言常常倾向于把程序写得紧凑，我们的 str_cpy 函数也可以写成

```
void str_cpy(s, t)
char *s, *t ;
{
    while ( (*s ++ = *t ++) ! = '\0') ;
}
```

这里把指针 s 和 t 的移动放到了循环测试部分，*t++的值是 t 加 1 之前所指的字符，后缀++在引用了字符值之后才改变 t 的指向，同样在 s 加 1 之前，字符存入原来 s 所指的位置，此字符同样与 '\0' 比较来控制循环，最后的结果是把字符串(连同 '\0')一起写到了 s 所指的地方。由于指针变量的加 1 运算放到了循环测试部分，所以循环体只需要一个空语句就可以了。

再仔细观察一下，str_cpy 还可以简化，由于 '\0' 的值就是 0。一个值不等于 0 时为真，则它自己就是非 0 值，而 C 语言中用非 0 值表示真，所以不等于 0 的判断是多余的，于是 str_cpy 可以写成

```
void str_cpy(s, t)
char *s, *t ;
{
    while (*s ++ = *t ++)
        ;
```

```
}
```

这种写法看起来表示的内容不太清楚，但确实有许多人就这样写程序。你应该熟悉这种写法，即使你自己不用。

7.4.3 几个字符指针的例子

在第六章中我们曾介绍过几个字符数组的例子，现在我们用指针来实现它们。

例 7.14

```
/* 返回 s 的长度 */
int str_len(s)
char *s;
{
    char *p;
    p = s;
    while (*p != '\0')
        p++;
    return(p - s);
}
```

这里首先把 s 赋给 p，也就是说，让 p 指向字符串的第一个字符，在 while 循环中逐个检查每个字符直到出现 '\0'，这时 p 指向了字符串的尾部，而 s 还指向字符串头，指针减法 p - s 就得到字符串中字符的个数，也就是字符串的长度。循环部分也可写成

```
while (*p)
    p++;
```

例 7.15

```
/* 连结 t 所指的字符串到 s 所指的字符串之后 */
void str_cat(s, t)
char *s, *t;
{
    while (*s != '\0')
        s++;
    while ((*s++ = *t++) != '\0')
        ;
}
```

第一个 while 循环将 s 移到第一个字符串的\0 处，第二个 whlie 循环将 t 所指的字符串复制到从原\0 开始的地方，原来 s 所指的字符串的\0 标志被覆盖，最终连结后的结果放在原 s 所指的存储空间中，如图 7-23 所示。

如想写得再简洁一些，两个 while 中的

```
!= '\0'
```

都可以省略。

例 7.16

```
/* 比较两个字符串 s 和 t，如果 s<t 返回负值，如果 s=t 返回 0， */
```

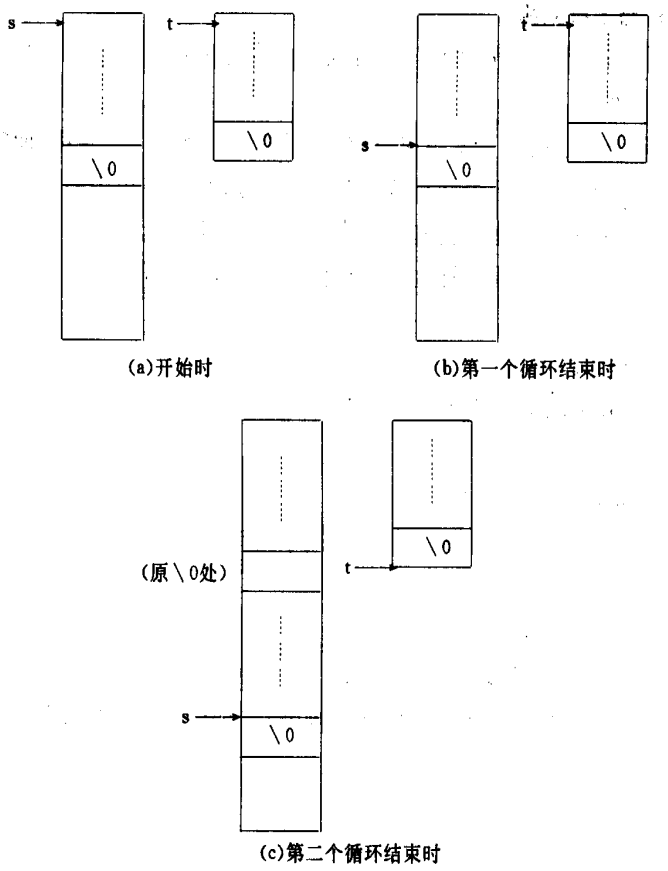


图7-23

```
/* 如果 s>t 返回正值 */
int str_cmp()
char *s, *t;
{
    whlie (*s == *t) {
        if (*s == '\0')
            return(0);
        s++;
        t++;
    }
    return(*s - *t);
}
```

while 循环的条件是 s 与 t 所指的对应的字符相等，如果两个字符串同时到达 '\0'，则表明它们相等，返回一个 0 值；如果从 while 循环中退出，就表明两个字符串中遇到了不相等的字符，这时执行

```
return(*s - *t);
```

返回两个不相等的字符的差值，如 s 所指的字符小于 t 所指的字符，就返回一个负值，否则返回一个正值。

7.4.4 字符指针与字符数组

虽然字符指针和字符数组都可以实现字符串的存储和运算，但它们并不相同，这里我们来讨论一下它们的差异。

字符数组由若干个字符型元素组成，每个元素中可以存放一个字符，而字符指针变量只能存放一个地址值。当我们用一个字符数组存放字符串时，字符串中的每一个字符占据字符数组中的一个元素，而用字符指针指向字符串时，只是把指针指向了存放字符串的那个存储空间的起始地址。一般来说，用指针比用数组操作字符串更方便，如你可以写

```
char *s;
s ="P.R.China";
```

使 s 指向字符串"P.R.China"的起址位置，但你不能写

```
char s[10];
s = "P.R.China";
```

因为 C 语言不允许向一个数组赋值，我们可以用字符串拷贝来完成这个操作，如

```
char s[10];
str_cpy(s, "P.R.China");
```

str_cpy 操作时，值是一个一个赋的。

由于指针可以灵活的移动，所以在处理字符串时会带来许多方便。

例 7.17

```
main()
{
    char *str = "P.R.China";
    str = str + 4;
    printf("%s\n", str);
}
```

运行结果

China

但对数组来说就没有这么方便了，你不能写

```
char str[] = {"P.R.China"};
str = str + 4;
printf("%s\n", str);
```

因为数组名表示数组的首地址，一但系统为数组分配了存储空间，你根本就不可能改变它，而且地址移动于理也说不通。当然你也可以直接写

```
char str[] = {"P.R.China"};
printf("%s\n", str + 4);
```

但不能象指针一样灵活地移动，总是不那么方便。

字符指针比字符数组方便，但使用时也要小心，因为它更容易出错，如可以写

```
char str[20];
scanf("%s", str);
```

而不应该写

```
char *str;
scanf("%s", str);
```

后者乍看起来也像是对的，但 str 在接收输入之前没有指向确定的位置，因此接收的数据是不可控制的，可能还会产生错误。

7.4.5 标准库中的字符处理函数

C 语言标准库中有许多用于字符串处理的库函数，虽然它们不是 C 语言中法定的部分，但由于所有系统都拥有它们，我们也可以把它们看做是 C 语言的一部分。下面介绍几个最常用的字符串函数。

1. strlen(s)

这里 s 是字符指针，这个函数返回\0 前的字符数，也就是返回字符串的长度。

2. strcmp(s1, s2)

这里 s1、s2 都是字符指针，函数按照字典顺序比较字符串 s1 和 s2，如果两者相等就返回 0，如果 s1<s2，就返回负值，如果 s1>s2 则返回正值。

3. strncmp(s1, s2, n)

与 strcmp 类似，只是最多比较前 n 个字符。

4. strcat(s1, s2)

s1 和 s2 都是字符指针，函数将 s2(包括\0)连结到 s1 的尾部，调用时应该保证 s1 有足够的空间来存放有关内容。

5. strcpy(s1, s2)

s1 和 s2 为字符指针，函数将 s2 的内容复制到 s1 中，调用时应该保证 s1 有足够的空间。

6. strncpy(s1, s2, n)

与 strcpy 类似，只是最多只复制 n 个字符，这时要注意，如果 s2 超过 n 个字符，s1 不以\0 结束。

7. index(s, c)

s 是字符指针，c 是字符，函数返回字符串 s 中字符 c 第一次出现的位置(指针)，如果 c 不在 s 中则返回空(用 NULL 表示)。

8. rinder(s, c)

与 index 类似，只是返回 c 最后一次出现的位置。

这些函数中前 3 个的类型是 int，如

```
int strlen(s)
```

后 5 个的函数类型是指向字符的指针，如

```
char *index(s, c)
```

关于函数类型为指针值的问题我们在下节讨论。

第五节 返回指针值的函数

一个函数总具有一定的类型，也就是它返回值的类型。函数可以返回整型、浮点型等，也可以返回一个指针值，这时函数的一般形式为

类型标识符 * 函数名(参数表)

如我们可以写一个函数

```
char *str_index(s1, s2)
```

它返回字符串 s1 中字符串 s2 第一次出现的位置。如果 s2 不出现在 s1 中则返回一个 NULL。

例 7.18

```
#define NULL 0
main()
{
    char *s1 = "This is a string";
    char *s2 = "is";
    char *str_index(), *p;
    p = str_index(s1, s2);
    if (p != NULL)
        printf("%s\n", p);
}
char *str_index(s1, s2)
char *s1, *s2;
{
    int n;
    n = strlen(s2);
    while (*s != '\0') {
        if (strncmp(s1, s2, n) == 0)
            return(s1);
        s1++;
    }
    return(NULL);
}
```

函数 str_index 中 while 循环每次将指针 s1 后移一个字符，然后判断从这个位置开始的 n 个字符是否与 s2 相当，也就是 s2 是否出现在 s1 中，如是，返回这时 s1 的值，指示出 s2 在 s1 中的位置，如果 s2 不在 s1 中出现，最后返回一个 NULL。

第六节　指针数组和指向指针的指针

7.6.1　指针数组

因为指针型变量也是变量，我们可以把多个指针变量组织起来构成一个指针数组。指针数组的定义形式为

类型标识符 * 数组名[数组长度]

如

```
int *p[10];
```

定义了一个指针数组，它的每一个元素(a[0]～a[9])都是一个指向整型值的指针变量。

为什么要用到指针数组呢？理由是可以用它来处理一组字符串。

考虑这样的问题：有一些国名，现想对它们排序，由于国名是不同长度的字符串，我们可以用一组指针来指向它们，这时采用指针数组是很方便的，如图 7-24 所示。

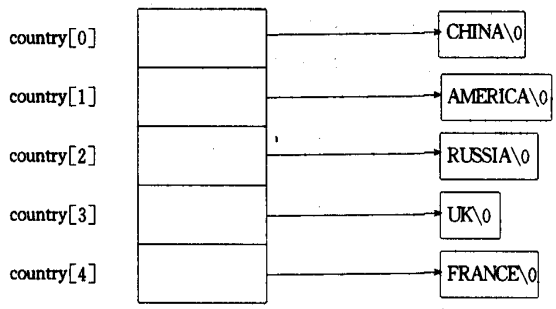


图7-24　指针数组

如果想对这些字符串排序，不必改动字符串的位置，只需要改动指针数组中各元素的指向即可，这样允许各字符串的长度不同，而且改变指针变量的值也比移动字符串花费的时间少。下面给出字符串排序的程序。

例 9.19 将国名按字母顺序输出。

```
main()
{
    void sort() ;
    void print() ;
    static char * country[]= {"CHINA",
                              "AMERICA",
                              "RUSSIA",
                              "UK",
                              "FRANCE"
};
    int n = 5 ;
    sort(country, n) ;
    print(country, n) ;
}
/ * 用 shell 方法排序 * /
void sort(v,n) ;
char  * v[] ;
int n ;
{
    int gap, i, j ;
    char  * temp ;
    for (gap = n / 2; gap > 0; gap /= 2)
```

```
        for (i = gap; i < n; i ++)
            for (j = i - gap; j >= 0; j -= gap)
                if (strcmp(v[j], v[j + gap] > 0) {
                    temp = v[j] ;
                    v[j] = v[j + gap] ;
                    v[j + gap] = temp ;
                }
}
void print(v, n)
char * v[] ;
int n ;
{
    int i ;
    for (i = 0; i < n; i ++)
        printf("%s\n", v[i]) ;
}
```

运行结果

```
AMERICA
CHINA
FRANCE
RUSSIA
UK
```

在 main 中定义了指针数组 country，它有 5 个元素，初始化时让它们分别指向"CHINA"、"AMERICA"、"RUSSIA"、"UK"和"FRANCE"5 个字符串，如图 7-24。sort 函数采用 shell 排序法，与前面整型序的主要不同之处是用

```
strcmp(v[i], v[j + gap]) > 0
```

代替了

```
v[j] > v[j + gap]
```

这是因为进行的是两个字符串的比较。当两个字符串顺序不对时，我们把指针做了对换(注意，对换的不是字符串本身)，执行完 sort 函数后，指针数组的指向如图 7-25 所示。

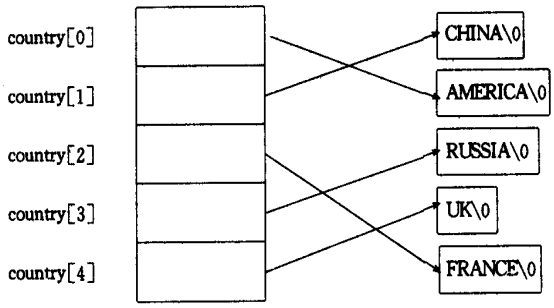


图7-25 字符串排序结果

字符串的位置没有改变，改变的只是指针数组各元素的指向。

7.6.2 指向指针的指针

指针可以用来指向另一个指针，如有定义

```
int i = 100 ;
int *i_p ;
int **i_p_p ;
```

i_p_p 就定义为一个指向指针的指针，执行两个赋值语句

```
i_p = &i ;
i_p_p = &i_p ;
```

后，i_p 指向变量 i，而 i_p_p 指向了 i_p，情况如图 7-26 所示。

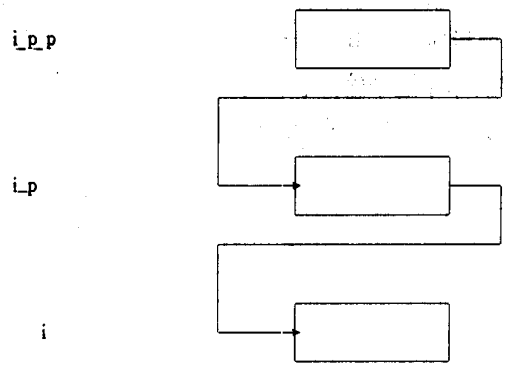


图7-26 指向指针的指针

这时要获取 i 的值 100，可用

```
i
```

这是直接访问，也可通过指向 i 的指针 i_p 间接访问，写做

```
*i_p
```

还可以通过 i_p_p 进行两次间接访问，写成

```
**i_p_p
```

i_p_p 是一个指向整型指针的指针，我们对它做一次间接访问运算

```
*i_p_p
```

得到的是一个指向整型的指针，也就是 i_p，再次做间接访问

```
*(*i_p_p)
```

就相当于

```
*i_p
```

也就得到了 i。

C 语言允许进行多重间接访问，也就是说定义变量时可以有多个 *，不过极少有用二次以上的间接访问，对于初学者来说，两次已经够复杂的了。

指向指针的指针主要用来操作指针数组，如例 7.19 中的 print 函数可以用指向指针的指针做为参数，然后用指向指针的指针操作字符串。

```
void print(v, n)
```

```
char * *v;
int n;
{
    char * *p;
    p = v;
    while (p < v + n) {
        printf("%s\n", *p);
        p++;
    }
}
```

这里 v 做为形参接收数组 country 的首地址，然后把 v 赋给 p，开始时 * p 就指向 country[0]，也就是说它是一个指向字符串的指针，以后 p 移动，* p 就指向后面的字符串。在 printf 中，* p 从指向的字符串中逐个输出字符值，也就相当于取指针的指针值。其效果与用 v[i]是一样的。这种用指针的指针操作的情况如图 7-27 所示。

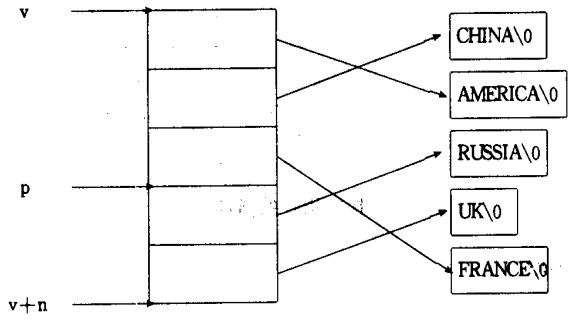


图7-27 用指回指针的指针操作指针数组

从图中我们可以看到 p 做为指向指针的指针。

7.6.3 命令行参数

指针数组(或指向指针的指针)的一个重要应用是作为主函数 main 的形式参数。在前面的程序中，main 总是不带参数的，写成

```
main()
```

实际上 main 也可以带参数。那么 main 的参数从哪里传递过来的？是操作系统。在执行一个程序时，我们总是在命令行上写

程序名 参数 1 参数 2 … 参数 n

这些命令行中的内容(包括程序名)都可以做为参数传给 main 函数，这种传给 main 的参数就是命令行参数。

main 函数要接收命令行传递过来的数据，就需要定义自己的形参。C 语言规定 main 可以带两个参数：第一个是整型量，一般用 argc 表示，它表明命令行参数的个数；第二个参数是指针数组或指向指针的指针，一般取名为 argv，用来存放各命令行参数的内容。于是带有参数的 main 函数可以写做

```
main(argc, argv)
```

```
int argc ;
char *argv[] ;
```

或

```
main(argc, argv)
int argc ;
char **argv ;
```

假定命令行的形式是

```
copy from to
```

则 main 接收的参数如图 7-28 所示。

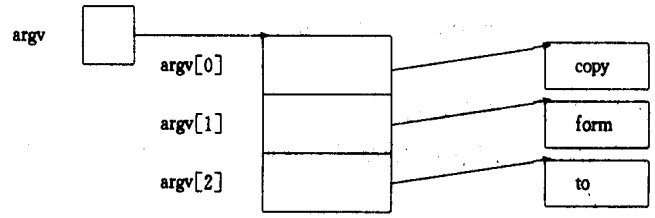


图7-28 命令行参数

即 argv[0]指向命令名"copy",argv[1]指向参数"from",argv[2]指向参数"to",实际上 argv[0]总是指向命令名的。

例 7.20 echo 程序。

程序 echo 的功能很简单,它把它后面的各命令行参数回送到输出行中,如给出命令

```
echo how do you do
```

输出为

```
how do you do
```

下面是实现它的程序。

```
/* 回送参数,版本 1 */
main(argc, argv)
int argc ;
char *argv[] ;
{
    int i ;
    for (i = 1; i < argc; i ++) {
        printf("%s", arg[i]) ;
        if (i < argc - 1)
            printf(" ") ;
        else
            printf("\n") ;
    }
}
```

这个程序逐个输出命令行参数,如果后面还有参数,就输出一个空格来分隔参数,如果

所有参数都输出完了，就输出一个换行符。

这个程序也可以写得紧凑些，如

```
/* 回送参数，版本 2 */
main(argc, argv)
int argc ;
char *argv[] ;
{
    int i ;
    for (i = 1; i < argc; i ++)
        printf("%s%c", argv[i], (i < argc - 1) ? ' ' : '\n') ;
}
```

这里用条件表达式代替了条件语句。

由于 argv 是指针数组(或指向指针的指针)，我们也可以只用指针来操作。

```
/* 回送参数，版本 3 */
main(argc, argv)
int argc ;
char **argv ;
{
    while (-- argc > 0)
    printf("%s%c", * ++ argv, (argc > 1) ? ' ' : '\n') ;
}
```

由于 argv 是指针，开始时它指向参数字符串数组的开始，先对 argv 加 1 使它第一次输出时就指向 argv[1]而不是 argv[0]。以后指针逐次移动，直到所有参数输出完毕(由 argc 控制)。

第七节　指向函数的指针

在 C 语言中，函数不是变量，但它同样占据一定的存储空间，也就是说函数也有自己的内存地址。这样就允许用一指针指向它，即指向函数的指针。如有一个函数

```
int max(x, y)
```

求两个整数较大者，同数组名一样，函数名表示函数存储区的起始地址，也就是说，简单的写

```
max
```

就得到一个函数的起始地址值，如图 7-29(a)所示，现定义一个指向函数的指针

```
int (*fnptr)() ;
```

这里表明 fnptr 是一个可以返回指向整型函数的指针，接着我们可以用赋值语句

```
fnptr = max ;
```

使 fnptr 指向函数 max，如图 7-29(b)所示。

在定义

```
int (*fnptr)() ;
```

中，*fnptr 两侧的括号是必须的，因为函数调用运算符()比间接运算符具有更高的优先级，

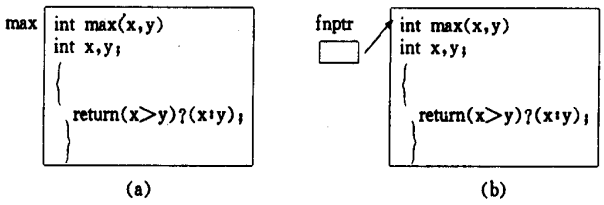


图7-29

如不写括号，就变成了

```
int *fnptr() ;
```

把 fnptr 说明成了一个函数，它返回一个指向整型的指针值，这在语法上是完全正确的，只是不是我们所希望的。

指针指向函数后，我们可以通过指针来调用函数，如我们可以写一个函数调用语句。

```
m = (*fnptr)(a, b) ;
```

这就相当于

```
m = max(a, b) ;
```

这样写没有什么奇怪的，其实，现在 fnptr 的值是函数 max 的起始地址，*fnptr 不就可以看作是 max 吗？括号只是为了保证正确的结合顺序。

例 7.21

```
main()
{
    int max() ;
    int (*fnptr)() ;
    int a, b, m ;
    a = 2 ;
    b = 5 ;
    fnptr = max ;
    m = (*fnptr)(a, b) ;
    printf("max=%d\n", m) ;
}
int max(x, y)
int x, y ;
{
    return( (x > y) ? x : y) ;
}
```

运行结果

```
max=5
```

main 函数中的说明。

```
int max() ;
```

在这种情况下是不能省略的，否则编译时会指出错误。

指向函数的指针是很有用的，其理由主要有二。其一，我们不能把函数本身做为参数传

递给另一个函数，但却可以把指向函数的指针做为参数传递，也就是说可以间接地把函数作为另一个函数的参数。

其二，我们不可以把整个函数存放在数组中统一管理，却可以用指向函数的指针构成数组。

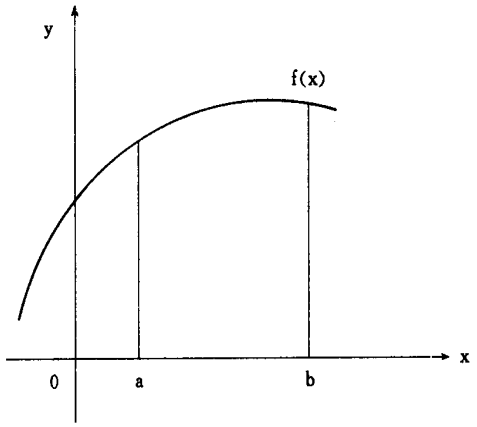
这样，有了指向函数的指针，我们就可以把函数象数据一样的处理，这在解决实际问题时会带来许多好处。

例 7.22　求定积分。

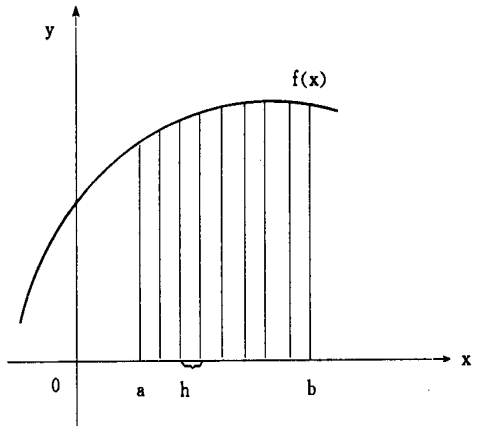
现考虑一下求定积分的问题。所谓函数 f(x)在[a,b]区间上的定积分

$$\int_b^a f(x)dx$$

其物理意义就是由曲线 f(x)，直线 x=a，x=b 及横轴所围成的面积。如图 7-30(a)所示。由于 f(x)一般为曲线，求曲线包围的面积不那么容易，我们可以考虑把 a、b 分成多个小区间，每个小区间中 f(x)可用一段直线近似代替，也就是说 f(x)可以近似地看作由许多折线组成，如图 7-30(b)所示。



(a)



(b)

图7-30　求定积分问题

这样曲线包围的面积就可以通过求多个梯形面积的和而近似求得，于是定积分

$$\int_a^b f(x)dx$$

可用下面公式求出

$$s=\frac{f(a)+f(a+h)}{2}\cdot h+\frac{f(a+h)+f(a+2h)}{2}\cdot h+\cdots+\frac{f(a+(n-1)\cdot h)+f(b)}{2}\cdot h$$

$$=\frac{h}{2}(f(a)+2f(a+h)+2f(a+2h)+\cdots 2f(a+(n-1)h)+f(b)$$

$$=\frac{f(a)+f(b)}{2}+f(a+h)+f(a+2h)+\cdots+f(a+(n-1)h)\cdot h$$

当 h 取值足够小时，s 值就很接近定积分的值。

有了求值公式，我们就可以对具体的函数来求其定积分了，如

$$\int_1^2 \sin(x)dx$$

可以写程序段

```
n = 100 ;
h = (2.0 - 1.0) / n ;
s = ( sin(1.0) + sin(2.0) ) / 2 ;
for (i = 1; i < n; i ++)
    s = s + sin(1.0 + i * h) ;
s = s * h ;
```

现在的问题用定积分近似公式可以对任何连续的 f(x)求值，当然也希望我们的程序能够适用于各种 f(x)。这就需要把函数作为参数传递给求定积分的函数。函数本身不能够作参数，我们可以利用指向函数的指针来间接地完成函数的传递。程序如下：

```
#include "math.h"
float interal(),f() ;
main()
{
    float a, b, y1, y2 ;
    y1 = interal(sin, 1.0, 2.0) ;
    y2 = interal(f, 0.0, 1.0) ;
    printf("y1=%0.2f\ny2=%0.2f\n", y1, y2) ;
}
float interal (fun, a, b)
float ( *fun)() ;
float a, b ;
{
    float s, h ;
    int n, i ;
    n = 100 ;
```

```
    h = (b - a) / n ;
    s = ( ( (*fun)(a) + (*fun)(b) ) / 2.0 ;
    for (i = 1; i < n; i ++)
        s = s + (*fun)(a + i * h) ;
    s = s * h ;
    return(s) ;
}
float f(x)
float x ;
{
    return(1 + x * x) ;
}
```

运行结果

y1=0.96

y2=1.33

这里第一次调求积分的函数求解库函数 sin 的定积分，第二次求解自己定义的函数 f(x)=1+x^2 的定积分。

最后说一点，指向函数的指针不能进行加减整数，两指针相减等运算，理由是显而易见的。

第八节　常见错误

1. 对指针变量赋予非指针值，如，

```
int i, *p ;
p = i ;
```

由于 i 是整型，而 p 是指向整型的指针，它们的类型并不相同，p 所要求的是一个指针值，即一个变量的地址，因此应该写做

```
p = &i ;
```

2. 使用指针之前没有让指针指向确定的存储区，如

```
char *str ;
scanf("%s", str) ;
```

这里 str 没有具体的指向，接收的数据是不可控制的，应该特别记住：指针不是数组！上面的语句可改为

```
char c[80], *str ;
str = c ;
scanf("%s", str) ;
```

3. 向字符数组赋字符串。

由于看到字符指针指向字符串的写法，如

```
char *str ;
```

```
str = "This is a string" ;
```

就以为字符数组也可以如此，写做

```
char s[80] ;
s = "This is a string" ;
```

这是错误的。C 语言不允许同时操作整个数组的数据，这时，你可用字符串拷贝函数：

```
strcpy(s, "This is a string") ;
```

4. 希望获得被调函数中的结果，却没有用指针，如

```
int a, b ;
a = 5 ;
b = 10 ;
swap(a, b) ;
printf("%d%d", a, b) ;
⋮
void swap(x, y)
int x, y ;
{
⋮
}
```

由于 C 语言的参数都是传值的，要想得到被调函数中的结果就需要使用指针，如

```
swap(&a, &b) ;
⋮
void swap(x, y)
int *x, *y ;
{
⋮
}
```

5. 指针做非法操作，如

```
int *l, *r, *x ;
x = (l + r) / 2 ;
```

由于 l 和 p 都是指针，它们不能相加，赋值可写成

```
x = l + (r - l) / 2 ;
```

6. 指针超越数组范围，如

```
int a[10], i, *p ;
p = a ;
for (i = 0; i < 10; i ++) {
    scanf("%d", p) ;
    p ++ ;
}
for (i = 0; i < 10; i ++) {
```

```
    printf("%d", *p);
    p++;
}
```

第一个 for 循环已使指针 p 移出了数组 a 的范围，第二个 for 循环操作时 p 始终处在数组 a 之外。使用指针操作数组元素时，应随时注意不要让指针越界。上面程序可以在两个 for 循环之间加上一句

```
p = a;
```

使 p 重新指向数组 a 的开始处。

7. 指向不同类型的指针一起操作，如

```
int *ipt;
float *fpt;
⋮
if (ipt - fpt > 0)
⋮
```

由于 fpt 和 ipt 指向不同类型的数据，它们之间根本不能一起参加运算，所以这是错误的。

8. 不同类型的指针赋值，如

```
int *p;
char *sf();
⋮
p = sf(…);
```

这里 p 是指向整型的指针，而 sf 返回指向 char 型的指针，这种赋值是不合理的，如果你真需要将返回结果送给 p，可用强迫类型转换，如

```
p = (*int)sf(…);
```

第八章　结构与其它数据类型

本章学习重点

本章介绍C语言的结构类型、结构数组和指向结构的指针，并在此基础上讨论动态数据结构的概念。此外，还将介绍联合类型、枚举类型及类型定义的概念。学习本章应熟练地掌握结构及指向结构的指针的概念，学会如何处理动态数据结构。

第一节　结构类型

8.1.1　结构的概念

在前面的各章中，我们介绍了C语言的基本数据类型：整型、浮点型、双精度浮点型、字符型和无值(void)型，也介绍了一种构造类型数据：数组。数组是由相同类型元素组成的构造类型。

在现实中，经常会遇到这样的问题，几个数据之间有着密切的联系，它们用来描述一个事物的几个方面，但它们并不属于同一类型。比如说职工的记录，这可由工作证号、姓名、性别、年龄、工资、家庭住址等数据项组成。这些数据项描述了一个职工的几个不同侧面。如果像图8-1分开来用独立的变量表示，很难看出这些数据有什么联系，处理起来也不方便。C语言提供了一种数据结构，它可以把不同类型数据(当然也可以是相同类型数据)，组织成一个整体，这就是结构类型。职工记录可用结构类型表示，如图8-2所示。

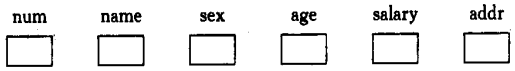


图8-1　用单个变量表示职工情况

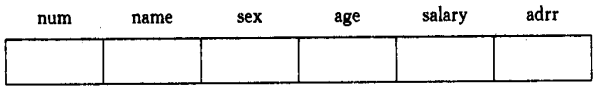


图8-2　结构类型表示的职工记录

8.1.2　结构类型和结构类型变量

前面讨论的职工记录，我们可以用一个结构类型来描述如下：

```
struct  staff {
    long  num ;
    char  name[20] ;
    char  sex ;
    int  age ;
    float  salary ;
    char  addr[30];
};
```

这里定义了一个结构类型 struct　staff，其中 struct 是关键字，用它来表示定义一个结构类型，staff 是结构名。我们看到，结构允许把不同类型的数据组织成一个整体。结构类型的一般定义为

```
struct　结构名{
    成员分量
};
```

花括号中的成员分量就是组成这个结构类型的各数据项，其定义形式为

类型标识符　成员分量名

不要忘记在括号后应该有个分号。

一但定义了结构类型，我们就可以用它来定义结构变量，就象定义其他类型的变量一样。如

```
struct　staff　stff1, stff2 ;
```

定义了两个 struct　staff 型的变量，它们在内存中的情况如图 8-3 所示。

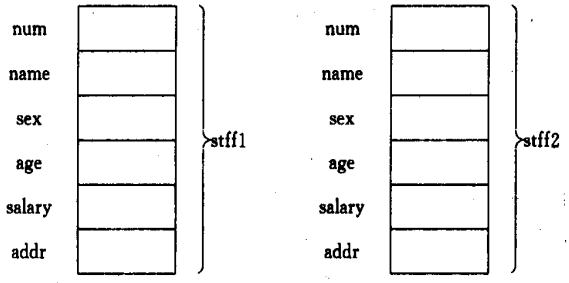


图 8-3　结构类型变量

定义结构类型变量时不能只写 struct，如

```
struct　stff1, stff2 ;
```

因为可用 struct 来定义多种结构类型。也不能省略关键字 struct 而只写结构名，如

```
staff　stff1, stff2 ;
```

这样看起来含义不很清楚。

我们可以先定义结构类型，再用结构类型定义结构变量，也可以在定义结构类型的同时定义结构变量，如

```
struct　staff {
    long　num ;
    char　name[20] ;
    char　sex ;
    int　age ;
    float　salary ;
    char　addr[30] ;
}stff1, stff2 ;
```

在定义 struct　staff 类型的同时定义了两个变量 stff1 和 stff2，这种定义的一般形式为

struct 结构名{

成员分量

}变量表列；

我们甚至可以不定义结构类型名而直接定义结构变量，如

```
struct {
    long  num ;
    char  name[20];
    char  sex ;
    int  age ;
    float  salary ;
    char  addr[30];
}stff1, stff2 ;
```

这种定义的一般形式为

struct {

成员分量

} 变量表列 ；

在定义结构类型时，成员分量可以是任意类型，如我们前面的定义中出现整型、字符型、浮点型和数组类型。C 语言也允许成员类型是结构类型，如 struct staff 的定义中，有一项是 age(年龄)。由于年龄总是随着时间变化，比较好的办法是用生日来代替年龄。生日是一个日期，日期由年月日组成，这又可以是一个结构，如

```
struct  date {
    int  year ;
    int  month ;
    int  day ;
}
```

于是 staff 可定义为

```
struct  staff {
    long  num ;
    char  name[20] ;
    char  sex ;
    struct  date  birthday ;
    float  salary ;
    char  addr[30] ;
} ;
```

这里用它来定义变量 struct staff stff ；其内存的情况如图 8.4。

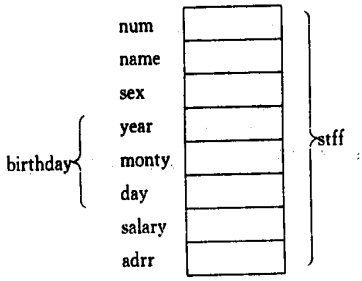


图 8-4

8.1.3 结构变量的引用

我们不能对结构变量的整体作操作，只能分别引用结构变量的各分量，引用的形式为

结构变量名.成员分量名

如

```
stff.num = 100001 ;
strcpy(stff.name,"zhang  shan") ;
```

这里的“.”叫做成员运算符，它是左结合的，具有最高的优先级。如果成员分量又是结构类型，必须一级一级地找到最低级分量，然后引用最低级分量，也就是只能对组成结构的基本类型进行操作，如

```
stff.birthday.year = 1960 ;
stff.birthday.month = 3 ;
stff.birthday.day = 25 ;
```

至于最低层成员分量所能进行的操作，由它们自己的类型决定。

8.1.4 结构变量的初始化

象数组一样，只有静态结构变量和外部结构变量才能够初始化，如

```
struct  staff {
    long  num ;
    char  name[20] ;
    char  sex ;
    int  age ;
    float  salary ;
    char  addr[30] ;
} ;
static  struct  staff  wang
        = {10001, "Wang  Jin", 'M', 30, 200.00, "123 ZhongHua Road"} ;
```

初始化时，将所提供的数据按照各分量的顺序排列，如果结构分量仍是结构，则按最低层类型提供数据，如

```
struct  date {
    int  year ;
    int  month ;
    int  day ;
} ;
```

```
struct  staff {
    long  num ;
    char  name[20] ;
    char  sex ;
    struct  date  birthday ;
    float  salary ;
    char  addr[30] ;
} ;
static struct  staff =
        {10002, "ZhuPing", 'F', 1963, 8, 30, 180.0, "321 Jilin  Road"} ;
```

下面通过一个简单的程序来看一下结构类型变量的定义、初始化和引用。

例 8.1

```
struct  staff {
    long  num ;
    char  name[20] ;
    char  sex ;
    int  age ;
    float  salary ;
    char  addr[30] ;
} ;
main()
{
    static  struct  staff  stff1
            = {1001, "Wang  Cheng", 'M', 40, 220.0, "1200 Beijing Road"} ;
    printf("NO:%ld\nName:%s\nSex:%c\n
    Age:%d\nSalary:%0.2f\nAddr:%s\n", stff1.num,
    stff1.name, stff1.age, stff1.salary, stff1.addr) ;
}
```

运行结果

```
NO.:1001
Name:Wang  Cheng
Sex:M
Age:40
Salary:220.00
Addr:1200 Beijing Road
```

在这个程序中，我们把结构类型定义在函数之外。实际中，在定义结构类型时，我们总是倾向把它写在所有函数的前面，这样做的好处是，在任何函数内都可以定义这种结构类型的变量。

第二节　结构数组

结构数组与其他数组的概念一样，只是它的元素是结构类型。例如

```
struct  staff {
    long  num ;
    char  name[20] ;
    char  sex ;
    int  age ;
    float  salary ;
    char  addr[30] ;
} ;
struct  staff  stff[10] ;
```

定义了一个有 10 个元素的结构数组，它的每个元素都是 struct　staff 类型，如图 8-5 所示。

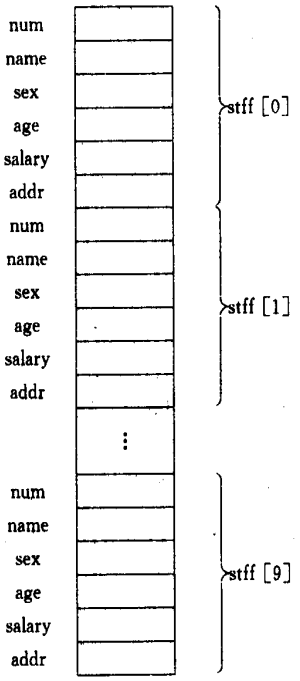


图 8-5　结构数组

引用结构数组的方式实际上就是引用数组和引用结构相结合，如

```
stff[0].num = 100001 ;
stff[5].salary = 300.0 ;
```

结构数组也可以初始化，方法是按一个一个数组元素提供数据。

例 8.2

```
struct  staff {
    char  name[20] ;
    float  salary ;
```

```
};
main()
{
    static  struct  staff  stff[] = {
                {"Wang  Cheng", 220.0},
                {"Li Lan", 195.0},
                {"Zhao  Qi", 315.0},
                {"Li  Fang", 280.0}} ;
    int  i ;
    for (i = 0; i < 4; i ++)
        if (stff[i].salary >= 200.0)
            printf("%-12s:%0.2f\n", stff[i].name, stff[i].salary) ;
}
```

运行结果

```
Wang  Cheng :220.00
Zhao  Qi    :315.00
Li  Fang    :280.00
```

第三节　指向结构的指针

8.3.1　指向结构类型数据的指针

一个结构类型的数据在内存中占据一定的存储空间，我们可以定义一个指向结构的指针来指向结构变量。

例 8.3

```
struct  staff {
    long  num ;
    char  name[20] ;
    float  salary ;
} ;
main()
{
    struct  staff  stff1;
    struct  staff  *stptr ; /*指向结构的指针*/
    stptr=&stff1 ; /*指针指向结构变量*/
    (*stptr).num = 1001 ;
    strcpy( (*stptr).name, "Wang  Cheng") ;
    (*stptr).salary = 220.0 ;
    printf("No. :%ld\n name:%s\nsalary:%.2f\n",
          (*stptr).num, (*stptr), name, (*stptr).salary) ;
```

```
}
```

运行结果

```
No:1001
name:Wang  Cheng
salary:220.00
```

程序中我们定义了一个指向结构 struct staff 类型的指针 stptr，赋值语句

```
stptr = &stff1 ;
```

使它指向结构变量 stff1，如图 8-6 所示。

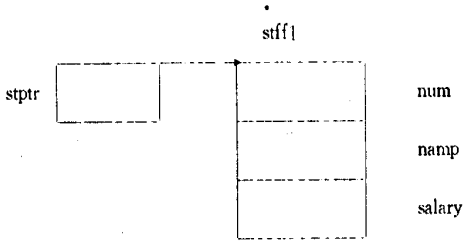


图8-6 指向结构的指针

这时的 * stptr 是 stptr 所指的变量本身，也就是 stff1，所以

```
( * stptr).num = 1001 ;
```

就相当于

```
stff1.num = 1001 ;
```

由于.的优先级高于 *，我们必须用括号来保证正确的结合。作为对照，下面给出不用指针的程序。

```
struct  staff {
    long  num ;
    char  name[20] ;
    float  salary ;
}
main()
{
    struct  staff  stff1 ;
    stff1.num = 1001 ;
    strcpy(stff1.name, "Wang  Cheng") ;
    stff1.salary = 220.0 ;
    printf("NO. :%ld\nname:%s\nsalary:%.2f\n",
        stff1.num, stff1.name, stff1.salary) ;
}
```

你可能觉得像(* stptr).num 这种形式不太方便，C 语言为了直观而方便地通过指向结构的指针来操作结构变量的成员，专门设置了一种指向运算符->，这样

```
( * stptr).num
```

就可以写成

stptr－＞num

其含义是 stptr 所指的那个结构变量的 num 分量。同样(＊stptr).name 和(＊stptr).salary 可以写成 stptr－＞name 和 stptr－＞salary。

指向运算符具有与．运算符相同的优先级，也就是最高的优先级。

同前面讨论的指向数组元素的指针一样，结构指针也经常用于操作结构数组。

例 8.4

```
struct  staff {
    long  num ;
    char  name[20] ;
    float  salary ;
} ;
main()
{
    static struct staff stff[] = {
        {1001, "Wang  Cheng", 220.0},
        {1002,"Li Lan", 195.0},
        {1003,"Zhao Qi", 315.0},
        {1004,"Lin  Fang", 280.0} } ;
    struct  staff  *stptr ;
    for (stptr = stff; stptr < stff + 4; stptr ++)
        if (stptr->salary > 220.0)
            printf("%ld:%s,%.2f\n", stptr->num,
                stptr->name, stptr->salary) ;
}
```

运行结果

```
1001:Wang Cheng,220.00
1003:Zhao  Qi,315.00
1004:Lin Fang,282.00
```

程序中我们定义了结构数组并使它初始化，for 循环中第一个赋值表达式 stptr=stff 将指针 stptr 指向了结构数组 stff 的起始位置，如图 8-7(a)所示。它也可以写成

stptr=&stff[0]

这与上一章讨论的指向数组元素的指针是一样的。由于这时 stptr 指的是 stff 中的第一个元素，所以 stptr－＞salary 这时就相当于 stff[0].salary，其他成员也是类似，第一次循环后，做操作

stptr ++

使指针指向数组中第二个元素 stff[1](与上一章讨论的一样，++总是使指针指向数组中的下一个元素，而不管数组元素是什么类型)，如图 8-7(b)所示。这时的 stptr－＞salary 就相当于 stff[1].salary 了。程序继续执行直到 stptr＞=stff+4。

在 C 语言中经常把指针加 1 或减 1 运算与指针的引用写在一起，如以前我们曾多次作

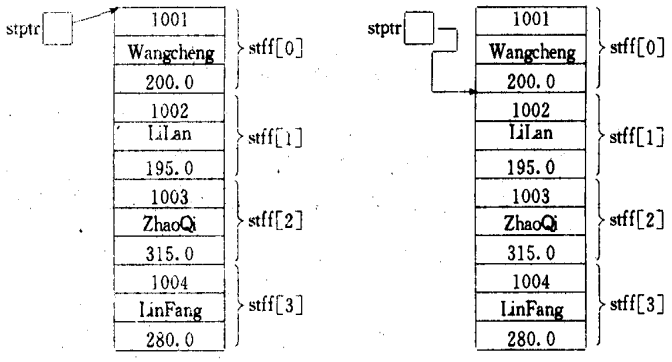


图8-7 用指针操作结构数组

过形如 * s ++的操作。如果你要想对结构指针作类似的操作,就要特别注意它的形式。现假定指针 p 指向数组 stff 的起始位置,则

```
++ p->num
```

表示的是把 stff[0]的 num 值加 1,因为->的优先级高于++,它等价于

```
++ (p->num)
```

如果你想先移动指针,就必须用括号把++和 p 括起来,如

```
(++ p)->num
```

这时它是先将指针后移一个元素,然后取那个元素的 num 值,也就是取 stff[1]的 num 值,如果写

```
(p ++)->num
```

则是先取 stff[0]的 num 值,然后移动指针 p 使其指向 stff[1]。

由此看来在使用结构指针时++(或--)的操作比较复杂,如果你怕弄不清楚,还是把引用与++(或--)分开写为好。如

```
n = (p ++)->num ;
```

不如写成

```
n = p->num ;
p ++ ;
```

更清楚些。

8.3.2 结构指针用作函数参数

有时候需要把一个结构型变量的值传给另一个函数,而早期的 C 不允许结构变量作为函数的参数,现在虽然 ANSI C 允许这样做,但一个结构中通常有较多的数据,传递是很费时的,所以最好是用指向结构的指针来作为参数间接地传递结构变量的值。

例 8.5

```
struct  staff {
    long  num ;
    thar  name[20];
    float  salary ;
};
main()
{
```

```
    void  print() ;
    static  struct  staff  wang = {1001,"Wang  Cheng",220.0} ;
    print(&wang) ;
}
void  print(stp)
struct  staff  *stp ;
{
    printf("%ld:%s,%.2f\n" ,stp->num, stp->name, stp->salary) ;
}
```

运行结果

1001:Wang Cheng,220.00

main 中取结构变量 wang 的地址作为参数传送给 print 中对应的形参 stp,这样 stp 就指向了结构变量 wang 的首地址,因而可以通过 stp 来引用这个结构变量的各成员分量了。

当然,用指向结构的指针作函数参数时,如果被调函数改变结构变量的值,也将会影响到主调函数,这里不再赘述。

第四节　引用自身的结构

8.4.1　指向自身结构的指针

前面讲过,一个结构的成员可以是任何类型,当然也可以是指针类型。这里介绍一种非常有用的指针类型成员,如

```
struct  listrec {
    int  a ;
     struct  listrec * p ;
} ;
```

这里定义的成员 p 是一个指针,它指向它所在的结构类型,这就意味着我们可以通过这种指针来引用自身类型的结构。使用这种引用自身的结构,我们可以构造出各种动态数据结构,比如说常用的链表和二叉树。

8.4.2　动态存储分配函数

要处理动态数据结构,就需要在执行程序的过程中动态地分配存储单元。C 语言提供了这方面的函数,这里我们只介绍两个最常用的。

1. malloc(size)

这个函数在内存中分配长度为 size 的连续存储空间,并返回这段存储空间的起始地址。如果分配不成功,则返回一个 NULL(空,其值为 0)。

2. free(ptr)

它释放由 ptr 所指的内存区,ptr 是调用 malloc 时的返回值。

有了这两个函数,我们就可以处理动态数据结构了。

8.4.3　链表

链表是一种动态分配存储单元的数据结构,它由一些结点组成,每个结点具有相同的类

型，并且每个结点都有一个可以指向其他结点的指针。图 8-8 给出了一个链表的形式，它由数据域和指针域组成。

图8-8 链表

第一个域是数据域，可用来存放所需要的数据，它可由多项数据组成。第二个域是指针域，用来存放指向相同类型结点的指针。链表中的最后一个结点的指针域值为空。一般链表还应有一个头指针，指向链表中的第一个结点，如图 8.8 中的 head。链表中组成一个结点的数据类型不同，应该把它定义为结构类型，最简单的结点只需要两个成员，如

```
struct  list {
    int  data ;
    struct  list * next ;
} ;
```

对于链表可以有多种操作，最常用的有

建立链表；

输出链表中的数据；

查找链表中的某个元素；

在链表中插入一个元素；

从链表中删除一个元素。

下面的例子给出它们的操作。

例 8.6　建立链表。

我们用一组给出的数据建立一个链表，并返回链表的头指针，这里只给出函数。

```
#define  NULL  0
struct  list {
    int  data ;
    struct  list  * next ;
} ;
struct  list  * creat_list(v, n)
int  v[], n ;
{
    void  * malloc() ; /* 有些 C 为 char * malloc() */
    struct list * head * p, * q ;
    int i ;
    if (n <= 0) ;
        return(NULL) ;
    /* 生成第一个结点 */
    q = (struct  list * )malloc(size  of(struct  list));
    q->data = v[0] ;
    head = q ; /* 确定表头的指向 */
```

```
    for (i = 1; i < n; i ++)'{
        /* 生成新结点 */
        p = (struct  list * )malloc(size  of(struct  list) ) ;
        p->data = v[i] ;
        q->next = p ; /* 连结上新结点 */
        q = p ;/* 移动指针 */
    }
    q->next = NULL ; /* 最后一个结点 */
    return(head) ;
}
```

这个函数接收从数组 v 传递过来的 n 个数据，建立一个有 n 个结点的链表。如果 n<=0 表示没有数据，返回一个 NULL 表示建立链表失败，否同就开始建立链表。语句

```
q = (struct list * )malloc(sizeof(struct list) ) ;
```

为一个结点分配存储空间，sizeof 是 C 中的一个单目运算符，它要求后跟一个数据类型，它的值是那个数据类型所占的字节数，如在 PC 机上 sizeof(int)的值为 2，sizeof(float)的值为 4。这样 sizeof(struct list)的值就是一个 struct list 结构所需要的字节数。把这个字节数做为参数，要求 malloc 分配一段能够放下一个结点的存储空间，malloc 函数返回一个可以适用于任何指针类型的 void 型指针(这是 ANSI C 标准，如果是 UNIX C 则返回一个指向字符型的指针)。由于 q 的类型是 struct list * ，所以使用类型强制转换运符(struct list *)使 q 指向新分配的结点，然后把第一个数值(v[0])送给这个结点，并使 head 指向这个结点。如图 8-9(a)所示。

for 循环将余下的数据放入链表中，操作过程是这样的：首先生成一个新结点，并用 p 指向它，接着把数据放入这个结点，然后语句

```
q->next = p ;
```

让前一个结点(q 所指的结点)的 next 域指向这个新结点，如图 8-9(b)，最后用语句

```
q = p ;
```

将指针 q 的指向移动一个结点，为下次做准备，如图 8-9(c)。循环外的赋值

```
q->next = NULL ;
```

将最后一个结点的 next 域置为空值，表示链表的尾部。语句

```
return(head) ;
```

返回这个链表的的头指针。

这个程序没有考虑当 malloc 分配存储空间失败的情况，你可以试着加上。

例 8.7　输出链表元素。

将链表中各结点的数据依次输出的操作很简单，首先要知道表头元素的地址，这可由 head 得到，然后顺着链表输出各结点中的数据，直到最后一个结点。程序如下：

```
void  print_list(head)
struct  list  * head ;
{
```

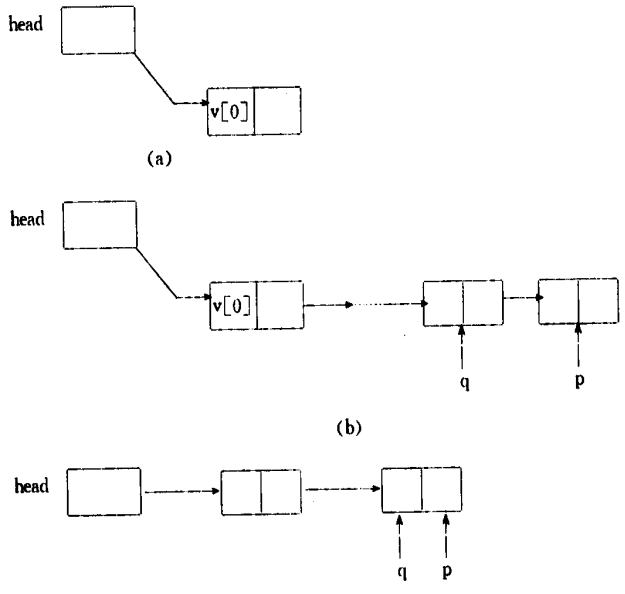


图8-9 链表生成

```
    struct  list   *p ;
    p = head ;
    while (p ! = NULL) {
        printf("%d\n", p->data) ;
        p = p->next ; /*移动*/
    }
}
```

首先将 p 指向 head，如果链表中没有内容(即只有一个空的头指针)当然就什么也不输出，否则就逐个输出各结点中的数据。语句

```
p = p->next ;
```

使 p 指向下一个结点。当输出完最后一个结点的数据后，p->next 的值是 NULL，把 NULL 赋给 p 也就结束了循环。这里用 NULL 作为链表的结束标志就象字符串用\0 作结束标志一样，使处理变得非常方便。NULL 实际上可以看成是一个值为 0 的指针，由于 C 语言保证不会认为 0 是一个合法的地址值，这样就可以用 NULL 来作特殊的标志(malloc 用返回 NULL 来表示出错)。

有了链表的建立和输出，我们可以编一段程序来测试它们。

例 8.8

```
#define  NULL  0
struct  list {
    int  data ;
    struct  list   *next ;
} ;
main()
```

```
{
    struct  list   *creat_list(), *head ;
    void   print_list() ;
    static  int   d[] = {1,2,3,4,5,6,7,8,9} ;
    head=creat_list(d, 9) ;
    print_list(head) ;
}
```

这个例子需要与函数 creat_list 和函数 print_list 连在一起，其运行结果为

```
1  2  3  4  5  6  7  8  9
```

例 8.9 查找链表中的某个元素。

给出一个数值，在链表中查找，如果找到就返回那个结点的地址，否则返回 NULL。程序如下：

```
struct  list   *   search(head, key)
struct  list   *   head ;
int   key ;
{
    struct  list   *p ;
    p = head ;
    while (p ! = NULL) {
        if (p->data == key)
            return(p) ;
        p = p->next ;
    }
    return(NULL) ;
}
```

例 8.10　在链表中插入一个元素。

这个操作是把一个结点插入在一个给定的结点之前。如果给定的结点不存在，就把新的结点放到链表的最后，返回值是指向新插入结点的指针。我们先来看一下一般的情况，如图 8-10(a)，p 指向存有 key 值结点的前一结点，r 指向要插入的结点，我们首先要做的是让 r 的 next 域指向有 key 的结点，如图 8-10(b)，这可用语句

```
r->next = p->next ;
```

来完成，然后再让 p 的 next 域指向 r，如图 8.10(c)，这可用语句

```
p->next = r ;
```

完成，这样就完成了插入工作。

现在来考虑一些特殊的情况。如果没有值为 key 的结点，r 所指的结点应放在最后，这时 p 就指向最后一个结点。情况如图 8-10(d)所示，现在我们要让 r 的 next 域为空，实际上由于 p 的 next 域这时为 NULL，所以也可以写成

```
r->next = p->next ;
```

然后让 p 的 next 域指向 r，如图 8.10(e)，仍然用

```
p->next = r ;
```

完成，这样看来，当 key 不存在时可同一般情况时一样处理。

再考虑一下，如果链表为空，也就是 head == NULL 时，我们需要做的是将 r 的 next 域置为空，而使 head 指向 r，如图 8.10(f)，这可用语句

```
r->next = NULL ;
head = r ;
```

完成。由于开始时 head 就为 NULL，也可以写成

```
r->next = head ;
head = r ;
```

最后再考虑如果第一个结点的值就是 key 的情况，这时则应让 r 的 next 域指向第一个结点，而这个结点原来是由 head 指向的，于是可用

```
r->next = head ;
```

实现，如图 8-10(g)，然后再让 head 指向 r，如图 8-10(h)，这可用语句

```
head = r ;
```

实现，也就是说与 head 等于 NULL 时一样。

有了这些讨论，我们就可以编程序了。程序如下：

```
struct list * insert(head, key, value)
struct list *head ; /*链表头指针*/
int key ; /*在 key 前插入*/
int value ; /*要插入的元素*/
{
    void * malloc() ; /*有些 C 用 char * malloc()*/
    struct list *p, *r ;
    r = (struct list *)malloc(sizeof(struct list)) ;
    r->data = value ;
    if (head == NULL || head->data == key) {
        r->next = head ;
        head = r ;
        return(r) ;
    }
    p = head ;
    while (p->next ! = NULL && p->next->data ! = key)
        p = p->next ;
    r->next = p->next ;
    p->next = r ;
    return(r) ;
}
```

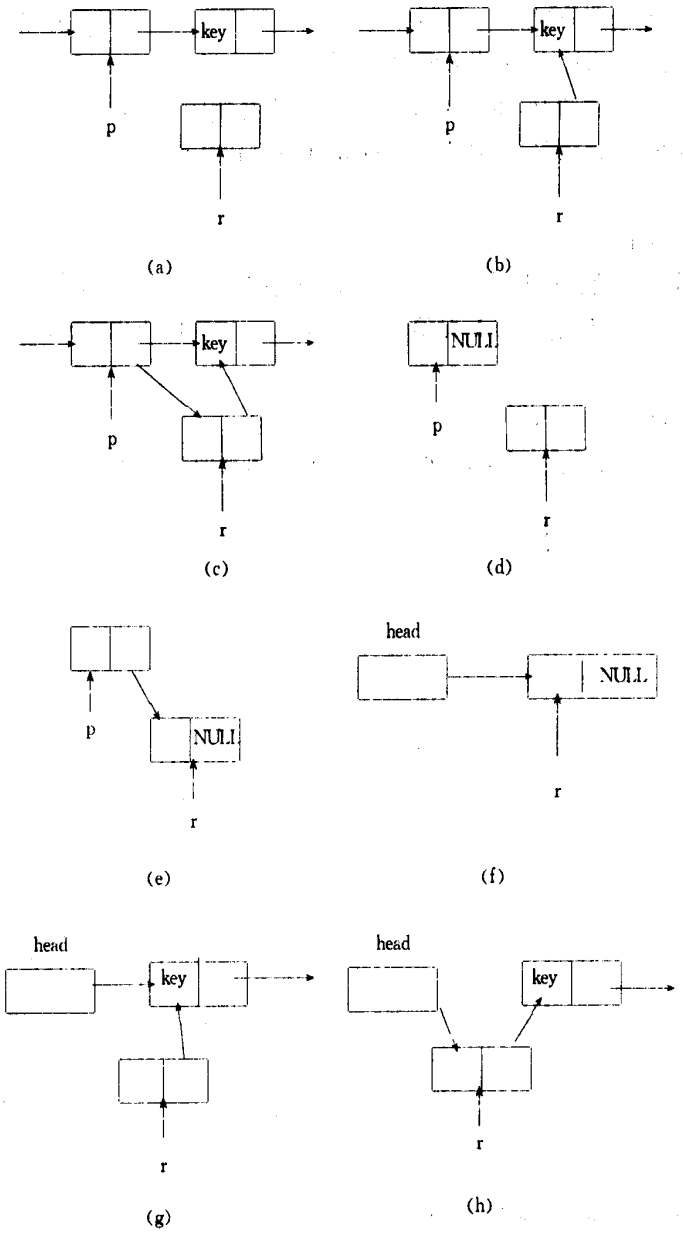


图8-10 链表插入

这里我们考虑一下逻辑表达式

head == NULL || head->data == key

如果 head == NULL，则 head->data 就无从谈起，而根据 || 运算的性质，一但前面的关系表达式为真，就不再求后面的关系表达式了，只有当 head==NULL 为假时才判断 head->data==key 是否为真，这样就保证了程序的正确性。如果将两个关系表达式位置颠倒，就可能出现错误了。用类似道理可以分析 while 循环中的逻辑表达式。

例 8.11 删除链表中的一个元素。

这个函数给出链表的头指针 head 和要删除的元素 key 做为参数，如果 key 存在就删除那个结点并返回头指针，如果 key 不存在则返回一个 NULL。

由于结点都是动态分配的，一但不再使用，我们就释放掉它所占的存储空间。为此，在这个问题中我们用两个指针，一个指向要删除的结点，一个指向前一个结点。

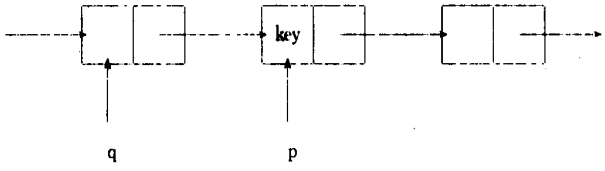
我们先考虑一般的情况，如图 8-11(a)所示，p 现在指向要删除的结点，q 指向 p 的前一个结点。删除的工作很简单，只要让 q 的 next 域指向 p 的 next 所指的结点即可，如图 8-11(b)。这可用赋值

```
q->next = p->next ;
```

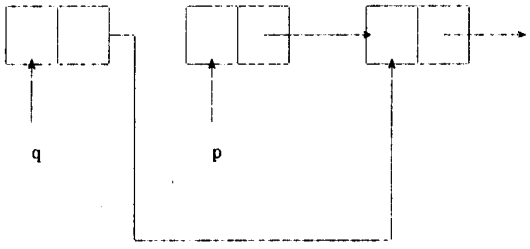
来完成，然后我们释放掉被删除的结点 free(p) 。如果 p 是最后一个结点，它的 next 域为 NULL，上面的赋值正好把 NULL 赋给了 q->next。比较特殊的情况是删除第一个结点，这时应做

```
head = head->next ;
```

如果 key 不存在，则只有 p 为 NULL 时才知道，这时可返回一个 NULL。



(a)



(b)

图8-11 链表元素删除

程序如下：

```
struct  list   *delete(head, key)
struct  list   *head ;
int   key ;
{
    void   free() ;
    struct  list   *p, *q ;
    p = head ;
    while (p ! = NULL && p->data ! = key) {
        q = p ;
        p = p->next ;
    }
    if (p == NULL) /*p 这时也可能等于 head */
```

```
        return(NULL) ;
    if (p == head) /*这时已保证链表非空*/
        head = head->next ; /*删除第一个结点*/
    else
        q->next = p->next ; /*删除一个结点*/
    free(p) ; /*释放 p 所指的结点所占的空间*/
    return(head) ;
}
```

8.4.4 二叉树

树也是一种很有用的数据结构，其中最常用的是二叉树。二叉树中每个结点最多有两个分支，它们被称为左子树和右子树，如图 8-12，其位置不能交换。

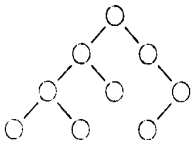


图 8-12 二叉树

二叉树有许多应用，这里只介绍二叉排序树。所谓二叉排序树具有这样的性质：任何一个结点的左子树中各结点的值都小于本结点的值，右子树中各结点的值都大于本结点的值。二叉排序树可以随着数据的增加而动态的建立。例如我们有数据

6 4 7 2 3 5 9 0 8

首先建立数据 6 的结点作为树根，如图 8.13(a)，再考虑下一个数据，由于 4<6，所以 4 做 6 的左子树，如图 8.13(b)，再考查数据 7，由于 7>6，让它做 6 的右子树，如图 8.13(c)，下面是数据 2，由于 2<6，所以走左子树，继续判断 2<4，故它做 4 的左子树，如图 8.13(d)，接着是数据 3，由于 3<6，走左子树，3<4，继续走左子树，现在 3>2，把 3 作 2 的右子树，如图 8.13(e)，再考虑数据 5，由于 5<6，走左子树。继续判断 5>4，作 4 的右子树，如图 8.12(f)，依次类推，最后得到这 9 个数据构成的二叉排序树，如图 8-12(i)。

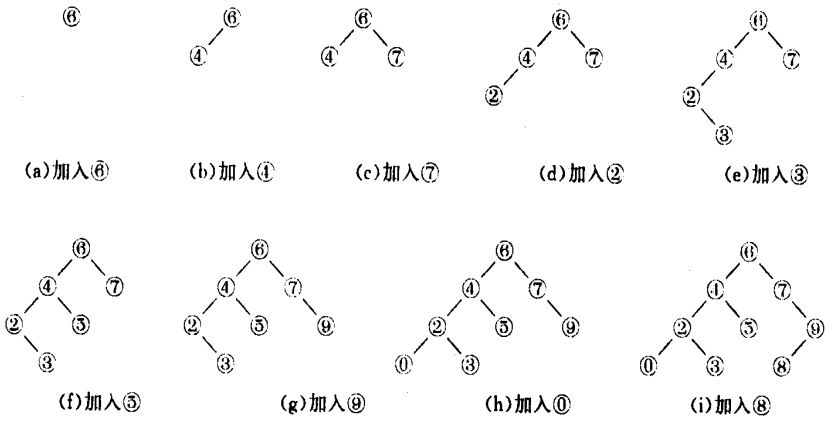


图 8-13 二叉排序树

按一定的方法访问二叉树中全部结点的过程叫二叉做树的遍历，常用的有先序遍历，中

序遍历和后序遍历，所谓中序遍历是这样的：从树根开始，如果左子树不为空就走左子树，左子树为空时访问这个结点，然后走一次右子树，以右子树为根再开始走左子树…，直到访问遍树中所有结点。图 8-13(i)中的二叉树，用中序遍历是这样的：从根结点开始，6 的左子树不为空，走左子树到 4；4 的左子树不为空，继续走左子树到 2；2 的左子树不为空，继续到 0；0 的左子树为空，访问 0；由于 0 结点没有右子树，下一个访问的结点是上面的 2，访问 2 后走右子树到 3，由于 3 没有左子树，访问它，3 也没有右子树，向上返回，2 已经访问过了，继续向上访问 4，然后访问 4 的右子树…，如此下去，得到结点的访问顺序为

0 2 3 4 5 6 7 8 9

也就是说二叉排序数按中序遍历得到的结果就是排好序的数列。现在我们用二叉排序树来解决一个实际问题。

例 8.12 用二叉排序树实现正文索引。

这个问题的要求是，读入一个正文，然后输出一份正文中出现的字及其每个字出现次数的表。字是以字母开头的字母数字串，现在我们用二叉排序树来实现这个问题，这时树中每个结点包含的数据有：

指向存放字的存储区的指针；

这个字出现的次数；

指向小于该字(字典顺序)的左子树的指针；

指向大于该字的右子树的指针；

用结构来描述就是

```
struct  tnode { /* 树结点 */
    char   *word ; /* 指向字的指针 */
    int  count ; /* 字出现的次数 */
    struct  tnode   *left ; /* 左子树指针 */
    strucl  tnode   *right ; /* 右子树指针 */
};
```

建立索引可借用二叉排序树的建立方法：确定一个读入的字是否已出现在树中，从根结点开始搜索，如新读入的字与结点中的字相同，就只增加计数，如新读入的字小于结点中的字就走左子树，否则走右子树，重复下去，如搜索不到存有这个字的结点，就说明这个字是第一次出现，这时为它建立一个结点，并放到树中。这个程序较长，我们先给出它再作分析。

```
#include  "stdio.h"
#define  MAXWORD  20
#define  LETTER 'a'  /* 字母标志 */
#define  DIGIT  '0' /* 数字标志 */
struct  tnode {
    char   *word ; /* 指向字的指针 */
    int  count ; /* 该字出现的次数 */
    struct  tnode *letf ; /* 左子树指针 */
    struct  tnode   *right ; /* 右子树指针 */
} ;
```

```
void  *malloc() ; /*有些C用 char  *mallo()*/
main()
{
    struct  tnode  *tree(), *root ;
    void  treeprint() ;
    char  wd[MAXWORD] ;
    int  t ;
    root=NULL ; /*初始化一棵树*/
    while( (t = getword(wd, MAXWORD) ) != EOF)
        if (t == LETTER) /*wd 中存有字*/
            root = tree(root,wd) ; /*加结点或次数到树中*/
    treeprint(root) ; /*输出树中的结点*/
}
struct  tnode  *tree(p, w)/*建立排序树*/
struct  tnode  *p ;
char  *w ;
{
    char  *strsave() ;
    int  cond ;
    if (p == NULL) { /*出现新字*/
        /*分配一个新结点*/
        p = (struct  tnode*)malloc(size  of(struct  tnode) ) ;
        p->word = strsave(w) ;
        p->count = 1 ;
        p->left = p->right = NULL ;
    }
    else  if( (cond = strcmp(w, p->word) ) == 0)
        p->count++ ; /*次数加1  */
    else  if(cond < 0)
        p->left = tree(p->left, w) ; /*走左子树*/
    else
        p->right = tree(p->right, w) ; /*走右子树*/
    return(p) ;
}
void  treeprint(p)
struct  tnode  *p ;
{
    if (p != NULL) {
        treeprint(p->left) ;
```

```
        printf("%4d %s\n", p->count, p->word) ;
        treeprint(p->right) ;
    }
}
char  *strsave(s)
char  *s ;
{
    char  *p ;
    if( (p =(char *)malloc(strlen(s) + 1) ! = NULL)
        strcpy(p, s) ;
    return(p) ;
}
int  getword(w, lim)
char  *w ;
int  lim ;
{
    int  c, t ;
    if (type(c = * w ++ = getchar() ) ! = LETTER) {
        *w = '\0' ;
        return(c) ;
    }
    while (-- lim > 0) {
        t = type(c = *w ++ = getchar() ) ;
        if (t ! = LETTER && t ! = DIGIT)
            break ;
    }
    *(w - 1) = '\0' ;
    return(LETTER) ;
}
int  type(c)
int  c ;
{
    if ( (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') )
        return(LETTER) ; /*是字母*/
    else  if  (c >= '0' && c <= '9')
        return(DIGIT) ; /*是数字*/
    else  return(c) ;
}
```

现在来分析一下这个程序：

主函数 main 不难理解，循环

while ((t = getword(wd, MAXWORD)) ! = EOF)

希望用 getword 获取一个字放在 wd 中，并返回一个标志给 t，如果 wd 中有一个字，则返回值为 LETTER，其他情况返回一个非字母的字符，当这个字符是文件结束符 EOF 时就结束循环。如果 wd 中已存放了一个字，就通过调用 tree 把它安装到树中。循环结束意味着统计完毕，最后用 treeprint 输出树中的各结点值。

函数 tree 是一个递归函数，用来建立二叉排序树，我们通过实际操作来看一下它的实现，当 main 函数第一次调用它时，由于 root 为 NULL，故形参 p==NULL，这时建立一个新结点，并把根结点的地址返回赋给 root。main 第二次调用 tree 时，p 不等于 NULL，则新字与 p 所指的结点(根)中的字作比较，如果它们相等，就把根结点计数加 1，如果它小于根中的字，就把它向左子树安装，否则向右子树安装。现在假定是向左子树安装，这时执行

p->left = tree(p->left, w) ;

以 p 的左子树为根，递归调用 tree，这次调用，由于传过来的参数 p(原来的 p->left)为 NULL，故生成新的结点，并返回结点的地址，递归调用结束，把这个结点的地址赋给了 p->left，也就是在 p 的左子树上安装了一个新结点，再有新结点依次类推，最终建立了一个二叉排序树。这个函数用递归形式显得比较清楚，如果不用递归，将是很麻烦的。

如果你理解递归，treeprint 函数就再简单不过了，这里可以看到递归表示的简洁性。

函数 strsave 的作用是开辟一段存储空间来保存一个字。由于树结点中不是定义一个存储字的数组，而是定义了一个指向字的指针 word，所以我们必须安排一定的地方来存放字，然后让指针指向它，strsave 就是完成这项工作的，它返回指向字的指针给 word。

函数 getword 用来从输入中获取一个字，如果获取的是字，就返回 LETTER 标志，否则就返回那个字符(这时它不会是 LETTER)。getword 中的条件表达式比较复杂，你可以拆成一部分一部分地看一下。

函数 type 用来确定输入字符的类型。

二叉树的应用是非常广泛的，如果你有兴趣，不妨找一本《数据结构》书看看。

第五节　联合

联合是又一种构造数据类型，它提供了一种可以把几种不同类型的数据存放于同一段内存的机制。系统程序设计中有时会有这种需求。如编译程序的符号表处理。现假定有三种不同类型的数据，分别是 char 型、int 型和 float 型，现在允许它们共用同一地址开始的内存单元。如图 8-14 所示，于是引出了联合的概念。

我们可以写

```
union  data {
    char  cval ;
    int  inal ;
    float  fval ;
} ;
```

这就定义了一种联合类型 union data，它允许三种类型的数据共用相同的存储单元。其中

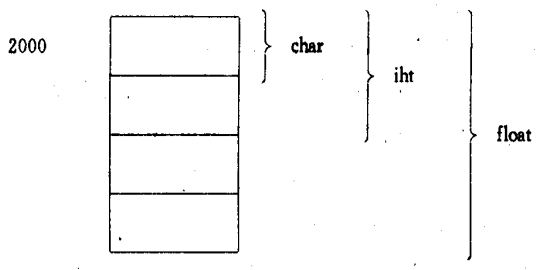


图8-14　不同类型数据共用相同的存储空间

union 是关键字，它表示 data 是一种联合类型名。

联合类型定义的一般形式为：

```
union  联合名 {
    成员表列
} ;
```

一旦定义了联合类型，我们就可以用它来定义变量了，如

```
union  data  d1,d2 ;
```

引用联合类型变量的方法与引用结构变量的形式是一样的，如

```
d1.ival = 10 ;
d2.fval = 2.5 ;
```

也可以定义并使用指向联合变量的指针，如

```
union  data  *p ;
p = &d1 ;
p->cval = 'a' ;
```

联合虽然形式上与结构类似，但它们有本质上的区别：结构中的成员是一个个分量，它们都有自己独有的存储单元，可以同时引用，而联合变量的各成员不是一个个分量，而相当于各种“身份”，它们不能同时存在。在任一时刻，一个联合型变量只能以各成员类型中的一个“身份”出现，也就是说，在任一时刻联合型变量中只能有一个成员起作用。

例 8.13

```
union  i_or_f {
    int  n ;
    float  x ;
} ;
main()
{
    union  i_or_f  var ;
    var.n = 100 ;
    printf("i:%d\n", var.n) ;
    var.x = 100.0 ;
    printf("f:%f\n", var.x) ;
}
```

```
i:100
f:100.000000
```

有关 union 类型，在 PC 机上的许多种 C 中有一个很常见的例子，这就是 union REGS。union REGS 的定义形式为：

```
union REGS {
    struct WORDREGS x;
    struct BYTEREGS h;
};
```

其中，struct WORDREGS 是一个描述 16 位寄存器的结构，它定义为

```
struct WORDREGS {
    unsigned int ax;
    unsigned int bx;
    unsighed int cx;
    unsigned int dx;
    unsigned int si;
    unsigned int di;
    unsigned int cflag;
};
```

struct BYTEREGS 是一个描述 8 位寄存器的结构，它定义为：

```
struct BYTEREGS {
    unsigned char al,ah;
    unsigned char bl,bh;
    unsigned char cl,ch;
    unsigned char dl,dh;
};
```

我们可以通过 x 来引用一个 16 位寄存器，也可以通过 h 来引用一个 8 位寄存器。如果你较深入地了解 PC 机上的 C 的话，就会发现许多实用程序都使用了 union REGS。

第六节 枚举

早期的 C 语言没有枚举类型，为了方便，后来的一些 C 引入了它。ANSI C 标准已正式增加了枚举类型。

如果一个变量只有几种可能的值，可以把它定义为枚举类型。所谓“枚举”，故名思义，就是把这种类型数据可取的值一一列举出来。一个枚举型变量取值仅限于列出值的范围。枚举数据类型通常的定义形式为

```
enum 枚举名{
    枚举元素表
};
```

例如，可以定义一个表示日期的枚举类型 weekday：

```
enum  weekday {sun, mon, tue, wed, thu, fri, sat} ;
```

这里enum是关键字,用于定义枚举类型,weekday是枚举类型的名字,枚举元素用标识符表示。

接下来我们可用enum weekday来定义变量,如

```
enum  weekday today, nextday ;
```

C语言也允许在定义枚举类型的同时定义枚举变量,如

```
enum  weekday {sun, mon, tue, wed, thu, fri, sat} today, nextday ;
```

这样,变量today和nextday就具有enum weekday类型,它们的取值只能是sun,mon…sat,如可有

```
today = sun ;
nextday = mon ;
if (today == sat)
    nextday = sun ;
```

C编译对枚举元素实际上按整型常量处理,当遇到枚举元素表列时,编译程序就把其中第一个标识符赋0值,第二、三…个标识符依次赋1,2…。因此,当枚举值赋值枚举变量时,该变量实际得到一个整数值。如

```
today = sun ;
```

是将值0赋给today,而不是将字符串"sun"赋给today。赋值后,

```
printf("%d", today) ;
```

的输出结果将是0。

你也可以在枚举类型定义时指定枚举元素的值,如

```
enum  weekday {sun = 7, mon = 1, tue, wed, thu, fri, sat} ;
```

这时sun的值是7,mon的值是1,而tue以后各元素的值,从mon的值开始,每次递增1,即tue的值为2,wed的值为3…。如果不写mon=1,则mon的值为8,tue的值为9…,依此类推。

由于枚举元素是常量,所以你不能在程序中对它们赋值,如sun = 0 ;mon = 1 ;这将产生错误。

既然枚举值就是整型值,那么它有什么存在的必要呢?至少有两个原因。一个是用标识符表示数值增加了程序的可读性,如

```
if (today == sat)
    nextday = sun ;
```

就比

```
if (today == 6)
    nextday = 0 ;
```

清楚多了;另一个更重要的原因是:它限制了变量取值的范围,如现在today只能取sun~sat中的值。

例8.14 荷兰国旗问题。

这是荷兰人Dijkstra提出的问题。荷兰国旗由红白蓝三色组成,现有N个桶,每个桶中放一个小球,小球是红的或白的或蓝的,要求是把这些小球重新排列,使红的排在前面,然后

是白的，最后是蓝的，并且规定每个桶只能看一次，当然要允许两个球交换，原问题还要求不使用数组，只能使用单个变量。现在我们把它改造一下，用数组来实现。

我们还是通过图来分析这个问题的解法。图 8-15 给出了一般时的数组中的情况。

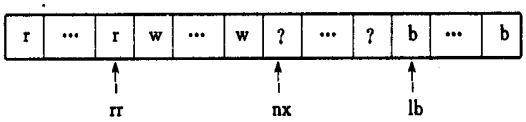


图8-15　荷兰国旗问题

这时数组元素已分为四部分：已知红色(r)，已知白色(w)，已知蓝色(b)和未检查(?)四类。三个指针表示最右边的红色(rr)，最左边的蓝色(lb)和要检查的下一个元素(nx)。

程序执行时，每次检查 nx 所指的值，如是白色(w)只需将 nx 加 1。如是红色(r)，可把它与 rr 的下一个元素互换，可先使 rr 加 1，然后互换 rr 和 nx 所指的元素，最后把 nx 加 1，因为换过来的是白色。

如果 nx 指的是蓝色，可以先把 lb 减 1，然后互换 nx 与 lb 位置的元素，把这个蓝色值放到新的 lb 处。

由于我们现在用数组处理这个问题，所以 rr，lb，nx 都表示为下标。很明显，在没有排序之前，rr 应在数组元素之前，lb 应在数组元素之后，可以表示为－1 和 n。

由于元素只取红、白、蓝三种值，所以可用枚举类型来实现它，程序如下：

```
#include "stdio.h"
enum  color {red, white, blue} ;
main()
{
    static  enum  color  flag[20] = {
    white,red,red,blue,white,red,blue,blue,white,blue,
    red,red,white,red,blue,white,blue,red,blue,white
    } ;
    enum  color  temp ;
    int  rr, lb, nx, i ;
    rr = -1 ;
    lb = 20 ;
    nx = 0 ;
    while (nx ! = lb) {
        if (flag[nx] == red) {
            rr ++ ;
            temp = flag[nx] ;
            flag[nx] = flag[rr] ;
            flag[rr] = temp ;
            nx ++ ;
        }
        else  if (flag[nx] == white)
```

```
            nx ++ ;
        else{ /* flag[nx] == blue  */
            lb -- ;
            temp = flag[nx] ;
            flag[nx] = flag[lb] ;
            flag[lb] = temp ;
        }
    }
    for(i = 0; i < 20; i ++)
        if (flag[i] = =red)
            putchar('r') ;
        else  if (flag[i] == white)
            putchar('w') ;
        else  /* flag[i] == blue */
            putchar('b') ;
    putchar('\n') ;
}
```

运行结果：rrrrrrrwwwwwwbbbbbbb

第七节　类型定义

除了可以直接使用基本数据类型名(如 char,int,long,float,double 等)和自己定义的结构、联合、枚举、指针类型名外,C 语言还允许用户自己定义新的类型名以代替已有的类型。这可用 typedef 来完成,如

```
typedef  int  INTEGER ;
typedef  float  REAL ;
```

指定用 INTEGER 代替 int,REAL 代替 float,于是定义

```
INTEGER  i, j, k ;
REAL  x, y, z ;
```

就相当于

```
int  i, j, k ;
float  x, y, z ;
```

通过这种方式我们可以把专门用于某一方面的变量用一个能表明它用途的类型定义,如

```
typedef  int  LENGTH ;
LENGTH  len, p ;
LENGTH  *lengths[] ;
```

这样一看就知道 len,p 和 lengths 都与长度有关,从而增加了程序的可读性。类似地,我们可以写

```
typedef  char  *STRING ;
```

命名 STRING 与 char *，也就是指向字符的指针等效。说明时可用如下形式

```
STRING  p, strsave() ;
```

typedef 也可以用来定义结构类型，如本章前面讨论的树结点，可作类型定义

```
typedef  struct  tnode {
    char  *word ;
    int  count ;
    struct  tnode *left ;
    struct  tnode *right ;
} TREENODE, *TREEPTR ;
```

这样，tree 函数的类型就可以写成

```
TREEPTR  tree() ;
```

一看就清楚 tree 返回一个指向树结点的指针，而分配一个结点空间的运算

```
p = (struct  tnode *)malloc(sizeof(struct  tnode)) ;
```

可以写成

```
p = (TREEPTR)malloc(sizeof(TREENODE)) ;
```

既清楚又易于书写。

必须指出的是，不论从任何意义上讲，typedef 都不产生新的类型，它只是给已有的类型增加新的类型名

使用 typedef 有两个主要原因。其一是可以使复杂的说明更易于理解，如用 TREEPTR 代替了 struct tnode *，使写出来的程序更清晰。

另一个原因是用 typedef 有利于程序的通用和移植。比如象 unsigned int 这样的类型，它与具体机器有关(PC 机上占两个字节，VAX－11 上占 4 个字节)，如在某种场合，我们需要固定地占两个字节，在 PC 机上我们写

```
typedef  unsigned  int  WORD ;
```

以后需要的场合都用 WORD 说明。当移到 VAX－11 上时，只需改写成

```
typedef  unsigned  short  WORD ;
```

程序中用 WORD 说明的地方都不需要改，这就减少了移植时的工作量。

最后说明一点，类型定义使用大写是为了醒目，不至于与常用的标识符混淆，并不是语法上必须的。

第八节　常见错误

1. 结构类型定义丢失分号，如

```
struct  date {
    int  year ;
    int  month ;
    int  day ;
}
```

这里右花号的后面少了一个分号，这种错误是因为习惯了复合语句的右花括号后不跟分号

造成的。这在编译时将指出错误，但不一定告诉你那里少了一个分号。这种情况同样可能出现在定义联合或枚举类型时。

2. 把结构名当做变量名。

```
struct  date {
    int  year ;
    int  mouth ;
    int  day ;
} ;
    ⋮
date. year = 1993 ;
date. month = 7 ;
date. day = 15 ;
```

这里的三个赋值是错误的，date 是结构类型名而不是变量名，只有变量才能够被赋值，而类型名只表示一种具体的类型，它不是变量，也没有自己的存储空间。对于上面的问题你可以先定义结构变量，然后引用结构变量的分量，如

```
struct  date  d ;
d. year = 1993 ;
d. month = 7 ;
d. day = 15 ;
```

3. 定义结构变量时丢失 struct，如

```
struct  date {
int  year ;
int  month ;
int  day ;
} ;
date  d ;
```

这里定义 d 时少写了 struct。C 语言要求，定义结构变量时不能只写结构类型名而不写 struct。

4. 定义结构变量时只写 struct，如

```
struct  x1, x2 ;
```

这是错误的，C 语言中的结构并不是只有一种类型，而是随你定义成各种结构类型，struct 只表明是结构类型，并没有指出是哪一种具体的结构类型。

5. 结构类型定义在一个函数内，而其他函数中要定义这种结构的变量，如

```
main()
{
    struct  date {
        int  year ;
        int  month ;
        int day ;
```

```
};
    ⋮
}
int  f()
{
    struct  date  d1, d2 ;
    ⋮
}
```

这在编译时会指出 f 中的 date 没有定义。要注意，结构名也是标识符，函数中定义的标识符的作用域是它所处的函数，在这个函数外是不可见的，所以 f 中 date 并没有定义。一般我们总倾向把各种类型的定义写在程序开始，所有函数之外，这样在每个函数中就都可以定义这些类型的变量了，如

```
struct  date {
    ⋮
};
main()
{
    struct  date  a, b ;
    ⋮
}
int  f()
{
    struct  date  d1, d2 ;
    ⋮
}
```

第九章　预处理程序

本章学习重点

本章介绍C语言编译预处理，包括宏替换、文件包含和条件编译。

C语言的一个重要特征是它有预处理功能。我们知道，一个高级语言源程序要在计算机上运行，必须先用编译程序将其翻译为机器语言。编译包括词法分析、语法分析、代码生成、代码优化等步骤，有时在编译之前还要做某些预处理工作，如去掉注释，变换格式等。C语言允许在源程序中包含预处理命令，在正式编译之前(词法分析之前)系统先对这些命令进行“预处理”，然后整个源程序再进行通常的编译处理。从语法上讲，这些预处理命令不是C语言的一部分，但使用它们却扩展了C语言程序设计的环境，可以简化程序开发过程，提高程序的可读性，也更有利于移植和调试C语言程序。

C语言提供的预处理命令主要有以下三种：

1. 宏替换
2. 文件包含
3. 条件编译

C语言的预处理命令都以#开头，以区别于一般C语句。

第一节　宏替换

9.1.1　不带参数的宏

不带参数的宏用指定标识符来代替一个字符串。其一般格式为

#define　标识符　字符串

如

```
#define  PI  3.1415926
```

用标识符PI来表示3.1415926，这叫做宏定义。一般来讲，标识符用大写字母，以便与程序中的变量名、函数名等区别开，看上去也更为醒目。这时的字符串是一个字符序列，除非是表示一个字符串常量，一般不用双引号括起。另外，宏不是语句，它不以分号结束，而以换行符结束。当宏比较长，在一行写不下时，可在行尾加上符号\表示下一行是这一行的继续，如

```
#define  MESSAGE  "the length  of  this  message\
is very  long"
```

一个宏的作用域是从定义处一直到本文件的结束。虽然不是必须的，但多数人习惯把所有的宏定义都写在文件的前面。

一个宏定义后，就可以在程序中使用了，如

```
#define  PI  3.1415926
```

```
double  area(radius)
double  radius ;
{
    return(PI * radius * radius) ;
}
```

这个函数返回圆的面积，如果不用宏可以写成

```
double  area(raius)
double  radius ;
{
    return(3.1415926 * radius * radius) ;
}
```

结果是完全一样的。

实际中预处理程序对宏的处理是，遇到宏名(标识符)就用其代表的字符串替换，即所谓宏替换，所以第一个程序在正式编译之前，系统就先预处理为第二个程序的形式。预处理是编译时由系统自动完成的，用户不必关心。从用户角度看，使用宏可以使程序具有更好的可读性。

程序中使用的常量一般都有一定的物理意义，人们很难从数字本身中看出这种意义。比如说，C 语言中没有专门的逻辑量，为了进行逻辑运算，规定表达式非 0 时为真(true)，给出的逻辑值为 1；表达式为 0 时为假(false)，给出的逻辑值为 0。这里的 1、0 与一般意义下的 1、0 形式上没有任何不同，不容易让人理解，有了宏，我们就可以很方便地解决这个问题了：

```
#  define  TRUE  1
#  define  FALSE  0
```

定义之后，当需要给出逻辑值真时，就使用 TRUE，当需要给出假时就用 FALSE。

再一个很有说服力的例子是处理函数的类型。有些 C 中没有关键字 void，当不需要函数返回值时，定义函数时不写出类型，但这实际上被系统默认为 int 型，含义不很明确。我们可以用宏定义

```
#define  VOID  int
```

自己定义一个 VOID 标识符，这样不返回值的函数，如

```
print_star()
{
    ⋮
}
```

就可以写成

```
VOID  print_star()
{
    ⋮
}
```

含义更明确了。

使用宏可以增加程序的可读性。它的另一个好处是使程序更易于修改，如

例 4.1 求一组数据中所有小于 20 的元素之和。

```
#define  SIZE  20
#define  LIMIT  20
main()
{
    int  array[SIZE], i, sum ;
    for (i = 0; i < SIZE; i ++)
        scanf("%d", array[i]) ;
    sum = 0 ;
    for (i = 0; i < SIZE; i ++)
        if(array[i] < LIMIT)
            sum = sum+array[i] ;
    printf("sum = %d\n", sum) ;
}
```

这里我们把数组的大小定义为 20,当需要在 30 个元素中求和时,只需要将宏定义

```
#define  SIZE  20
```

改成

```
#define  SIZE  30
```

即可,程序的其他部分可以不动,如果我们不使用宏定义,而是在需要的地方都写上数字本身,如:

```
main()
{
    int  array[20], i, sum ;
    for (i = 0; i < 20; i ++)
        scanf("%d", array[i]) ;
    sum = 0 ;
    for (i = 0; i < 20 ;i ++)
        if (array[i] < 20)
            sum = sum + array[i] ;
    printf("sum = %d\n", sum) ;
}
```

这当以后需要改成在 30 个元素中求和时,就要仔细地查找程序各部分的 20,而且还要注意,并不是所有 20 都代表元素的个数,如 if (array[i] < 20)中的 20 就不是元素个数,你不能将它也改成 30。

在定义宏时,还可以用到以前定义的宏,如

```
#define  PI  3.1415926
#define  RADIUS  2.0
#define  AREA  PI * RADIUS * RADIUS
```

这里定义 AREA 时用到了另两个宏 PI 和 RADIUS。

最后还要特别强调两点：

1. 宏定义以换行结束，不要用分号结束，如果 PI 定义为

```
#define  PI  3.1415926 ;
```

则

```
return(PI * radius * radius) ;
```

经替换后变成

```
return(3.1415926 ; * radius * radius) ;
```

一看就知道是错误的。

2. 如果双引号内出现与宏名相同的字符串，则这个字符串不被替代，如

例 9.2

```
#define  HELLO "how  are  you"
main()
{
    printf(HELLO) ;
    printf("\nHELLO\n") ;
}
```

运行结果

```
how  are  you
HELLO
```

第一个 HELLO 由于不在双引号内，预处理时它被 how are you 替换，printf 语句的形式为

```
printf("how  are  you") ;
```

第二个 HELLO 在双引号中出现，在预处理时它不被替换，所以第二个 printf 输出 HELLO。

9.1.2 带参数的宏

宏也可以带参数，其定义的一般形式为：

#define 标识符(参数表) 字符串

其中标识符是宏名，字符串中包含括号内指定的参数，称做宏扩展，如

```
#define  area(r)  (3.1415926 * (r) * (r))
```

这里 r 作为宏 area 的参数，定义之后可在程序中使用它，如

```
main()
{
    printf("%f\n", area(2.0)) ;
}
```

这个程序在编译预处理时被替换，替换时用给出的实参代替对应的形参，替换后的程序是

```
main()
{
    printf("%f\n", 3.1415926 * 2.0 * 2.0) ;
}
```

你可能注意到了，在这里宏的名字用的是小写字母，由于带参数的宏从形式上看很像函

数，为了统一，多数人在定义带参数的宏时喜欢使用小写字母。

宏也可以带多个参数，如求两个数中较大者，我们可以定义宏

```
#define  max(x, y) ( (x) > (y) ? (x) : (y) )
```

如果在程序中出现语句

```
m = (a, b) ;
```

则预处理时替换为

```
m = max( (a) > (b) ? (a) : (b) ) ;
```

如果语句为

```
m = max(p + q, r + s) ;
```

则被替换为

```
m = ( (p + q) > (r + s) ? (p + q) : (r + s) ) ;
```

定义带参数的宏也可以用到已定义的宏，如

```
#define  max(x, y)  ( (x) > (y) ? (x) : (y) )
#define  max_three(x, y ,z)  (max(max(x, y) ), (z) )
```

使用带参数宏编写程序时，要注意两点：

1. 在宏定义中宏名和括起参数的左圆括号之间不能有空格。

如果定义为

```
#define  area  (r) (3.1415926 * (r) * (r) )
```

则语句

```
printf("%f\n", area(2.0) ) ;
```

被替换为

```
printf("%f\n",(r) 3.1415926 * (r) * (r)(2.0) ) ;
```

显然是不对的。因为预处理程序把 area 认做是一个不带参数的宏，只是简单地把其后所跟字符串原样照搬到 area 处。

2. 整个宏扩展及各参数要用括号括起，就像前面给出的例子那样。如果不用括号括起会出现什么呢？我们来定义一个求平方的宏。

```
#define  square(x)  x * x
```

如果有语句

```
a = square(n + 1) ;
```

则预处理时被替换成

```
a = n + 1 * n + 1 ;
```

结果是把 2 * n + 1 赋给了 a，显然这不是所期望的

```
a = (n + 1) * (n + 1) ;
```

宏扩展中最外层的括号也是必要的。如果不要最外层的括号，宏定义写成

```
#define  square(x)  (x) * (x)
```

现用下面的语句调用

```
printf("%d\n", 27 / square(3) ) ;
```

你可以先猜猜看输出结果是什么。是 3 吗？让我们替换一下，变成

```
printf("%\n", 27 / (3) * (3) ) ;
```

由于*、/同级，并按从左到右顺序计算，所以表达式 27 / (3) * (3)的执行顺序是 27 除以3，得到值 9，然后用 9 乘以 3，得到 27，这不是我们所期望的值。现在你大概明白为什么要用括号括起整个宏扩展了吧。如果 square 定义为

```
#define  square(x)  ((x) * (x))
```

则

```
printf("%d\n", 27 / square(3));
```

被替换成

```
printf("%d\n", 27 / ((3) * (3)));
```

将输出值 3，这才是我们所要求的。

9.1.3 宏与函数

从形式上看，带参数的宏调用和函数调用没有什么区别，而且确实在一些场合下它们产生同样的结果，如

例 9.3

程序 1：

```
#define  max(x,y) ((x) > (y) ? (x) : (y))
main()
{
    printf("%d\n", max(2, 5));
}
```

程序 2：

```
main()
{
    printf("%d\n", max(2, 5));
}
int  max(x, y)
int  x, y;
{
    return((x > y) ? x : y);
}
```

这两个程序的主函数 main 是完全一样的，调用宏和调用函数得到相同的运行结果，都是 5。

但这种情况并不总是成立的。如

例 9.4

程序 1：

```
main()
{
    int  i;
    i = 1;
    while (i <= 10)
```

```
        printf("%d\n", square(i ++) ) ;
}
int square(x)
int  x ;
{
    return(x * x) ;
}
```

执行的结果为

```
1
4
9
16
25
36
49
64
81
100
```

程序 2

```
#define  square(x)  ( (x) * (x) )
main()
{
    int  i ;
    i = 1 ;
    while (i <= 10)
        printf("%d\n",square(i ++) ) ;
}
```

运行结果为

```
2
12
30
56
90
```

这是怎么回事？我们还是看看预处理时替换的结果。

```
square(i ++)
```

被替换成

```
(i ++) * (i ++)
```

这样一来你就知道为什么会出现这种结果了。

```
2 = 1 * 2
```

12 = 3 * 4

30 = 5 * 6

56 = 7 * 8

90 = 9 * 10

5 次循环后，i 值超过了 10，循环也就结束了。

你必须始终记住，C 预处理程序并不认识 C，它不做任何 C 语法的检查，更不管程序的意思，只是机械地按照宏定义把宏调用替换为对应的字符串。

在程序中为什么要使用带参数的宏呢？理由是：使用宏比函数调用更快。因为宏在真正编译之前已被相应地替换，在执行时，不必打断调用程序的运行，也没有参数的传递，而使用函数调用，当执行到有函数调用的语句时，主调函数要把参数传给被调函数，同时把控制权转给被调函数，被调函数运行完后再返回主调函数，这些都需要许多时间上的额外开销，当这种调用很频繁时（比如说在一个循环次数较多的循环体中），程序执行的速度就比较慢了。另一方面，由于预处理时每遇到宏调用就将对应的内容替换过来，因而程序较长，所占空间较多。反观函数，不管调用多少次，它的程序都那么长，所占空间比较少。

第二节　文件包含

所谓“文件包含”是指一个源文件可以将另一个源文件的全部内容包含到自己的文件中。文件包含命令的一般形式是

#include　"文件名"

它的作用是用指定文件的全部内容来代替本文件中的这一行。图 9-1 给出了文件包含的解释。

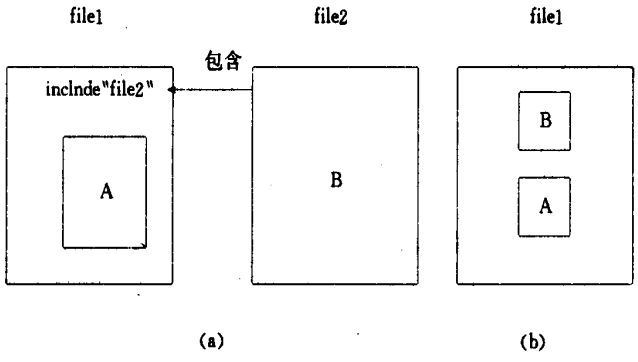


图9-1　文件包含

图 9-1(a)中，file1 中含有命令

#include　"file2"

在编译时，预处理程序找到文件 file2，然后用 file2 的全部内容代替命令

#include　"file2"

这一行，预处理后 file1 的情况如图 9-1(b)所示。

在程序中，文件包含是很有用的，写程序时，经常要定义一些符号常量（不带参数的宏）

和带参数的宏，而一个大程序通常是放在几个文件中，如果每个文件都重新定义这些符号常量和带参数的宏，那是很麻烦的，而且可能带来不一致的错误。为了解决这个问题，可以把它们单独放在一个文件中，其他文件在开头都使用文件包含命令将它们包含进来。例如，源程序中要使用下列常量：

```
#define  BUFSIZ  512
#define  EOF    -1
#define  NULL    0
#define  TAB   '\t'
#define  TRUE    1
#define  FALSE   0
#define  YES     1
#define  NO      0
```

现在它们放在一个文件中，取名 defs.h。如果有一个程序的两个源文件用到这些定义，就可以分别在两个文件的开头写上

```
#include  "defs.h"
```

习惯上，我们总是把这些包括宏定义的文件叫做头文件，后缀用.h(.h 表示 header，这个习惯来自 UNIX 操作系统)。实际上，任何一种 C 都提供了大量的.h 文件，其中包含符号常量，带参数的宏的定义，库函数的类型说明及系统中固定使用的结构或联合类型的定义，比如说我们用过的 stdio.h 和 math.h。当然你也可以对任何名字的文件(如.c 文件，即 C 语言源程序文件)使用文件包含命令。

文件包含命令中的文件名，也可以用尖括号括起来，如

```
#include <文件名>
```

两者的区别是：用尖括号时，系统只在“标准”目录中寻找文件，如 UNIX 系统在/usr/include 目录中查找文件，而用双引号括起文件时，系统先在当前目录中找，如找不到，再到其他目录中查找，最后查找所谓“标准”目录。

第三节　条件编译

一般情况下，源程序中所有的语句都参加编译。但有时也希望根据一定的条件去编译源文件的不同部分，这就是条件编译。条件编译使得同一源程序在不同的编译条件下得到不同的目标代码。商业软件公司总是使用条件编译来提供和维护某一程序的多个顾客版本。

条件编译有几种常用的形式，现分别介绍如下：

1.
```
#ifdef  标识符
        程序段 1
#else
        程序段 2
#endif
```

这种形式的含义是：如果标识符已被#define 行定义，则编译程序段 1，否则编译程序段 2。其中#else 及程序段 2 可以没有。

例 9.5

```
#define  LI
main()
{
#ifdef  LI
    printf("Hello,LI\n") ;
#else
    printf("Hello,everyone\n") ;
#endif
}
```

运行结果输出

```
Hello,LI
```

如果从程序中去掉

```
#define  LI
```

则输出结果为

```
Hello,everyone
```

2. #ifndef 标识符

```
        程序段 1
  #else
        程序段 2
  #endif
```

这与前一种形式的区别仅在于，如果标识符没在#define 行定义就编译程序段 1，如果定义了就编译程序段 2。

3. #if 表达式 1

```
        程序段 1
  #elif 表达式 2
        程序段 2
        ⋮
  #else
       程序段 n
  #endif
```

它的作用是，如果表达式 1 为真就编译程序段 1，否则如果表达式 2 为真就编译程序段 2，…，如果各表达式都不为真就编译程序段 n，例如下面程序段利用 ACTIVE_COUNTRY 定义货币的名字。

```
#define  USA  0
#define  ENGLAND  1
#define  FRANCE  2
#define  ACTIVE_COUNTRY  USA
#if  ACTIVE_COUNTRY == USA
```

```
    char   * cuurrency = "dollar" ;
#elif  ACTIVE_COUNTRY == ENGLAND
    char   * currency = "pound" ;
#else
    char   * currency = "franc" ;
#endif
```

除了上面三种形式中出现的条件编译命令外,C 语言中还有一种条件编译命令也较常用,它是

```
#undef
```

其功能是使前面定义的标识符变为未定义。

为什么要用条件编译呢? 主要原因是便于程序的移植。如在 PC 机上,最常用的 C 有 Turbo C 和 MS C,两者在实现上有一些不同处。如果我们希望自己的源程序能够适应这种差异,可以在它们形式不同的地方写上

```
#ifdef  TURBOC
    ⋮          /* Turbo C 独有的内容 */
#endif
#ifdef  MSC
    ⋮          /* MS C 独有的内容 */
#endif
```

如果希望这个程序在 Trubo C 环境下编译运行,可在程序前面写上

```
#define  TURBOC
```

如果希望生成 MSC 版本,就在程序前面写上

```
#define  MS C
```

这样一个源程序只要修改一句就可以适应两种 C 编译,商业软件公司的软件经常都是这样编写的。

使用条件编译的另一个主要原因是便于调试程序,一般当我们开始写一个程序时难免出错,为了便于检查错误,通常是加一些输出来跟踪中间结果。当然你可以先写一些 printf (或其他函数)语句输出,在程序调试完毕后再删除这些 printf 语句。但一个更好的办法是把它们写到条件编译部分去,如

```
#define  DEBUG
    ⋮
#ifdef  DEBUG
    printf(……) ;/* 临时结果 */
#endif
```

在调试时,由于定义了 DEBUG,所以临时结果被输出,这有利于监示程序是否正确。一但程序完成,这些输出就变得多余了,这时可以从程序中去掉 #define DEBUG,然后再编译,临时结果就不再输出。一般有经验的程序员总是用这种方法来书写和调试程序。

第四节　常见错误

1. 定义宏时在末尾加分号，如

```
#define  PI  3.1415926
#define  RADIUS  2.0
#define  AREA  PI * RADIUS * RADIUS ;
```

这种错误往往发生在宏比较长的情况下，特别是以表达式的形式出现或带参数时发生得较多，这是因为习惯了在语句的后面写分号所致。不过你应该清楚，宏定义不是语句，而是预处理命令，它用换行符结束。如果写了分号，在宏替换时，将连同分号一起被替换过去。如

```
return(AREA) ;
```

将变成

```
return (3.1415926 * 2.0 * 2.0 ;) ;
```

这将造成错误。

2. 丢失"#"，如

```
include  "stdio.h"
```

C 语言中的预处理命令都是以#开头的，这样预处理程序才能很方便地找到它们。丢失"#"将引起错误，这种错误往往发生在初学者只写一两个预处理命令时。正确的写法应是

```
#include  "stdio.h"
```

3. 定义字符串常量没有用引号，如

```
#define  HELLO  how are you
main()
{
    printf(HELLO) ;
}
```

这在编译时将会指出 how 没定义一类的错误。由于宏替换时只是把字符串原样照搬过来.这样

```
printf(HELLO) ;
```

就变成了

```
printf(how  are  you) ;
```

当然是错误的。所以，如果是定义一个字符串常量，就应该照字符串常量的形式用双引号括起字符序列，如

```
#define  HELLO  "how  are  you"
```

4. 宏扩展的整体或参数没用括号括起，如

```
#define  square(x)  x * x
```

或

```
#define  square  (x) * (x)
```

这在语句为

```
k = i / square(j + 1) ;
```

时被替换为

```
k = i / j + 1 * j + 1 ;
```

或

```
k = i / (j + 1) * (j + 1) ;
```

正确的写法是：

```
#define  square  ( (x) + (x) )
```

第十章　位运算

本章学习重点

本章介绍位运算的概念、位运算符及位域。学习本章应掌握用位运算编程的方法，为处理一些与硬件密切相关的操作打下基础。

第一节　位运算的概念

前面我们曾多次说过，C 语言是为了编写系统程序的需要而设计的。指针就是一个很好的例子，它是程序员控制和存取内存的有效工具。编写系统程序还要经常与计算机的“位”打交道，这方面，C 语言再次显示了它比其他一些通用语言(如 PASCAL)更优越的地方，它为程序员提供了大量位操作运算符。

所谓位运算是指二进制位的运算，在系统软件中这是经常遇到的问题。为了更好地了解位运算，我们先介绍一些有关的知识。

10.1.1　字节与位

在前面的讨论中，我们多次提到过字节的概念。对于大多数计算机系统而言，字节是由 8 个更小的叫做“位”的单位组成的，一个位可以取 0 或 1 中的某个值。如有一字节，它现具有值

01010100

字节的最右边一位是“最低有效位”或“最低位”，字节最左边一位叫做“最高有效位”或“最高位”。如果我们把这个 0,1 组成的串看做是二进制整数，则它的值对应十进制为

$2^2+2^4+2^6=4+16=64=84$

在计算机中各种类型的数据都占据一个或几个字节，如在 PC 机上，int 型数据占两个字节并用最左边一位(最高位)作为数的符号位。

10.1.2　补码

计算机中都有加法器，但再设专门的减法器就显得没有必要了。我们希望用加的方法来实现减，这就可用补码形式表示的数来完成。计算机中一般采用初码形式表示数。

补码的思想来自模运算，有关模运算的最直观的例子是钟表，如果现在时针指向 7，我们顺时针拨动时针到 3，共需拨动 8 格，如果逆时针拨动到 3，则需拨 4 格，也就是说在模 12 的情况下

$7+8$

与

$7-3$

的值都是 4，在数学中可表示为

$7+8=7-3=4 \quad (\mathrm{mod}12)$

这里 8 与 4 称为模 12 时的互补数。这样看来，减法的运算就可以用与它的互补数相加的方法求得。补码是这样定义的：

如果一个数是正数，则就用它本身的二进制形式表示。如果是负，则符号位为 1，其余各位为其二进制绝对值各位取反，再在最低位加 1 的值，如

+13　　为 00001101

−13　　为 11110011

它是这样求得的：

首先符号位为 1，然后其余各位取+13 二进制表示数的反，即 0 变 1，1 变 0，如

0001101

1110010　　各位取反

最后再在末尾加一个 1，　如

```
  11110010
+         1
------------
  11110011
```

这就是−13 的补码表示形式，由于正数时符号位为 0，负数时符号位为 1，所以可连符号位一起求反。

现在再来考虑一下将一个负数的补码转换为十进制数的问题：这可对补码的各位取反，再转换成十进制数，加上负号，最后再减 1。如：

11101101

一看最高位是 1，表示一个负数，现先对各位求反，变成

00010010

转换为十进制

$2^1+2^4=2+16=18$

加上负号再减 1，最后得到−19。

有了补码，减法运算就可以通过加上它的补码来实现。如

13 − 12

可以先求−12 的补码，12 的二进制表示为

00001100

则−12 的补码为

```
  11110011
+         1
------------
  11110100
```

然后与 13 的补码相加

```
    00001101
+   11110100
------------
1   00000001
```

舍去溢出的最高一位(模运算),结果为

00000001

正是我们所期望的值。

这里我们只是给出用一个字节表示数的情况。如果用多个字节表示一个数,则只有一个符号位,它处在最高字节的最高位,如

10011101　01110010

↑

符号位

以补码形式在一个字节中存放数据其取值范围是－128～127,如用两个字节存放一个补码值,取值范围是－32768～32767,这正是PC机上int型数据的取值范围。

如果我们不考虑符号位,如无符号整型unsigned　int型,则最高位也用来表示数值,它的取值范围是0～65535。

第二节　位运算符

现在我们来讨论C的位运算符,表10.1给出了C语言的全部位运算符。

表10.1　位运算符

符号	含义
&	按位与
|	按位或
∧	按位异或
～	取反
<<	左移
>>	右移

表中除了取反运算～是单目运算符外,其余都是双目运算符。位运算符只能用于整型量(包括int,char,short,long,unsigned等),而不能用于操作浮点型量(float和double)。

下面分别介绍各运算符。

1. 按位与(&)运算符

当两个数值作按位与运算时,是对其值的二进制表示形式逐位进行比较操作,如果两值对应位上都是1则这位取值1,否则取值0,其真值表见表10.2。

表10.2　按位与

b1	b2	b1&b2
0	0	0
0	1	0
1	0	0
1	1	1

如有两个数11和6进行按位与运算

11 & 6

则可表示为

```
    00001011
&   00000110
--------------
    00000010
```

结果为2。在程序中我们可以写语句

```
a1 = 11 ;
a2 = 6 ;
a3 = a1 & a2 ;
```

按位与运算经常用于取一个数据的若干位，如有一个整型值i，它占两个字节共16位二进制，数值为：

```
0101100100111001
```

现想取其低字节8位，可用表达式

```
i & 255
```

求得，其求值形式为

```
  0 1 0 1 1 0 0 1 0 0 1 1 1 0 0 1
& 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
-----------------------------------

  0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 1
```

也就是说把需要取出的部分用全1按位与，而其他位用全0按位与。这种方法适合于选取任何位(可以不连续)。

当在位运算中使用常数时，用八进制或十六进制表示是比较方便的。因为一位八进制正好对应三位二进制，而一位十六进制正好对应四位二进制数。于是取i的低八位可写成

```
i & 0377
```

而取i的高字节可写成

```
i & 0177400
```

2. 按位或(|)运算符

按位或时，对应位上只要有一个1，这位就取1，两个都是0时才取0值。其真值表见表10.3。

表10.3 按值或

b1	b2	b1|b2
0	0	0
0	1	1
1	0	1
1	1	1

如有表达式

```
075 | 022
```

按位或运算为

```
    00111101
|   00010010
------------

    00111111
```

结果为077,也就是十进制63。

按位或常用来对数据的某些位置1,如

```
i | 0377
```

将i的低字节8位置1,而高八位保留原值。

3.按位异或(^)运算符

异或的规则是:如果对应位上的值相同则这位取0值,如两个值不同则这位值取1。表10..4给出其真值。

表10.4 按位异或

b1	b2	b1^b2
0	0	0
0	1	1
1	0	1
1	1	0

如有表达式

```
066 ^ 035
```

相应的求值形式为

```
    00110110
^   00011101
------------

    00101011
```

结果为053,也就是十进制43。

异或运算一个有趣的性质是:任何位与自身异或都产生0。这很正常,因为在每一位上自己与自己当然相同。在汇编语言中这个事实被用来迅速把某个值置0。

异或的另一个有趣的性质是:一个值与任何其他值连续做两次异或操作结果都恢复为原来的值。我们通过例子来看一下这种情况。

```
054 ^ 061 ^ 061
    00101100 <- 054
^   00110001 <- 061
------------

    00011101
^   00110001 <- 061
------------

    00101100
```

结果仍是054。这实际上可以通过任何一位的情况分析出来。由于连续两次相同的异

或，原来的值或者不变，或者变化两次，而在只有两种值的情况下，变化两次也相当于不变。异或的这种性质常被用于图形显示，当一个画面的一部分被另一个画面遮盖时，就可以对两个画面的数据做异或操作，当那个外层画面移出时，只需要再做一次相同的异或操作，这样不必把被遮盖的画面的数据保存起来，在执行时可获得较快的速度。许多动画软件就是这样做的。

4. 取反(～)运算符

取反运算符是位运算符中唯一的一个单目运算符，它用来对数据的各二进制位求反，即1变成0，0变成1。如

～053

的结果是0324。说明如下：

053 －＞ 00101011

～ 053 －＞ 11010100

当我们不知道被处理量的准确位数时，取反运算符是很有用的。如我们希望把一个整数量w的最低位置成0，在PC机上可以写

w & 0177776

由于0177776的二进制值为

1111111111111110

则作按位与运算时，将保留w中的前15位原值，只把最低位置0，然而在一个整数为32位的计算机(如VAX－11)上显然用

w & 0177776

就不行了，因为w的高16位也都被置成了0！这怎么办呢？当然可以写

w & 037777777776

但这样一来，两种机器中的程序就不通用了。比较好的应该是与具体机器无关的写法，这时我们可以借助求反运算符写成

w & ～1

由于1的二进制表示是最后一位为1，前面各位都是0，求反后，得到的是前面各位都是1，最低一位为0的值。这对于两字节整数而言，前面有15个1，而对占四字节的整数来讲，前面共有31个1，不管怎样都可以使w保留前面各位的原值，而只把最后一位置0。这里表达式

w & ～1

不需要加括号，因为～的优先级在位运算符中是最高的。

5. 左移(＜＜)运算符

左移运算符需要两个操作数，写在前面的是要移动的数据，写在后面的是要移动的位数。如

a ＜＜ 2

表示a左移两位。如a的值为0163，则左移两位的结果如下：

a　　：01110011

a ＜＜ 2：11001100

也就是说移出的高位自动丢失，而低位总是移入0值。

当没有高位丢失时，左移一位就相当于乘 2 运算，如数据

013

左移一次后变成

026

左移两次变成

054

由于移位运算比乘法运算快得多，有些 C 编译就把乘 2 的幂运算转换左移运算。

6. 右移(>>)运算符

右移运算的形式与左移类似，也是运算符前面为要移动的量，后面为移动的位数。如 a >>2 表示 a 右移两位。移位时，移出右端的低位自动丢失，而移入的高位则与符号有关，如果是无符号数，高位移入 0，如 a 定义为 unsigned int 型时，a 的值为

1011011001110101

右移两位后变成

0010110110011101

如果是有符号数，对于正数(最高位为 0)移入的总是 0，而当数为负值(最高位为 1)时，则取决于具体的系统，如果是所谓带符号数扩展的系统就移入 1，如果是不带符号扩展的就移入 0。

以上是 C 语言的 6 种位运算符，下面我们再作两点说明：

1. 双目运算符可与赋值号一起构成赋值运算符。

像 C 语言中所有双目算术运算符一样，通过加上一个赋值号，双目位运算符也可构成赋值运算符，如：

```
w1 &= 037 ;
```

等价于

```
w1 = w1 & 037 ;
```

而

```
w2 <<= 3 ;
```

等价于

```
w2 = w2 << 3 ;
```

2. 不同长度的数据也可以进行位运算。

如果两个不同长度的数据，如 long 型和 short 型数据进行位运算时，系统把运算数按右对齐，如果较短的数是无符号数，左端就用 0 补齐，如果较短的数是有符号数，就按最高位的值补齐，也就是说为正数时用 0 补足，为负数时用 1 补足。

第三节　位运算应用举例

例 10.1 写一个函数 getbits(x,p,n)，它返回 x 中从右起第 p 值开始的 n 位字段，这里要求 p 和 n 是合理的正值，最右端为第 0 位。如图 10-1(a)所示。

要做这个工作首先我们要把从 p 开始的 n 位移动最右端，如图 10-1(b)所示，这可用表达式

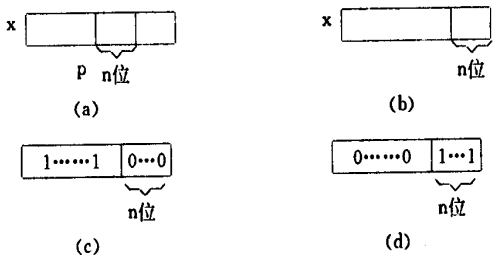


图10-1

x >> (p + 1 − n)

完成。

现在我们需要一个后 n 位全 1，而前面各位全 0 的值，这可通过下面的表达式求得。

~(~0 << n)

首先产生一个全 1 的数(~0)，然后左移 n 位使最后 n 位为 0，而前面全为 1，如图 10-1c)所示。最后再对它求反就得到了最后 n 位为 1，前面全为 0 的值，如图 10-1(d)所示。

现在可以把这两部分(图 10-1(b)和图 10-1(d))做一个与运算，就取出了所需要的 n 位。程序如下：

```
unsigned getbits(x, p, n)
unsigned x, p, n ;
{
    return (x >> (p + 1 − n) & ~(~0 << n) ) ;
}
```

例 10.2 写一个函数，计算所使用的系统中整型数据的长度。

这个程序可以这样考虑，先生成一个全 1 的值，然后通过左移，把各位的 1 一个个的移出，而右端这时会自动的补 0，当所有 1 都移出后，数就变成了 0，而移动的次数就表示整型数据的长度。程序如下：

```
int intlen()
{
    int i ;
    unsigned v ;
    v = ~0 ; /* 置成全 1 */
    i = 0 ;
    while (v ! = 0) {
        v = v << 1 ;
        i++ ;
    }
    return (i) ;
}
```

第四节 位 段

编程时经常遇到标识一个对象的特征的问题。如在编译程序中处理符号表时，要记录程序中出现的每个标识符是否是关键字，是否是外部的，是否是静态的等。在编写设备驱动程序时总要用到设备状态情况，如磁盘机要有是否出错，是否准备好以便传输数据，是否可以响应中断，当前正在读、写等特征。我们可以用整型量或字符型量来描述这些特征，而实际上这些特征往往用一个逻辑量就可以标识，也就是说，每个特征往往用一位二进制就可以了，而且对于设备状态来说，通常是放在设备状态寄存器中，设备寄存器总是用一位(或几位)表示一个特征。因此状态一般处在一个字节(或字)中，占据一位或几位。这样一个便于系统程序设计的语言，如 C 就应当提供处理字节(或字)中某一位或几位的功能，对于这种要求你已经了解能够通过位运算来完成，例如 10.1 就可以取出字中的任何位。但那毕竟不是很方便，为解决此类问题，C 语言提供了一种直接访问字位的手段，这就是位段。

C 语言用于访问位段的方法是基于结构的，位段实际上是一种特殊的结构，它以位为单位定义类型的长度。定义的一般形式为：

```
struct 结构名 {
    类型 标识符 :长度 1 ;
    类型 标识符 :长度 2 ;
    ..............
    类型 标识符 :长度 n ;
} ;
```

定义中的每一行就是一个位段，位段的类型只能是 unsigned 型(语法上还允许是 int 型，但这没有实际意义)。如处理符号表时，记录标识符信息可以写成

```
struct iden_flags {
    unsigned is_keyword : 1 ;
    unsigned is_extern : 1 ;
    unsigned is_static : 1 ;
} flags ;
```

这就定义了有三个位段的变量 flags，需要注意的是在存储单元中位段分配方向是因机器而异的，既可以从左到右分配(如图 10-2(a)所示)，也可以从右到左分配(如图 10-2(b)所示)。

位段的长度不必都是 1，如一个指令字由操作码和两个操作数组成，设操作码占 8 位，操作数各占 4 位，就可以表示为

```
struct instruction {
    unsigned opcode : 8 ;
    unsigned oper1 : 4 ;
    unsigned oper2 : 4 ;
} insword ;
```

对位段中数据的引用与引用普通结构的分量在形式上是一样的。如：

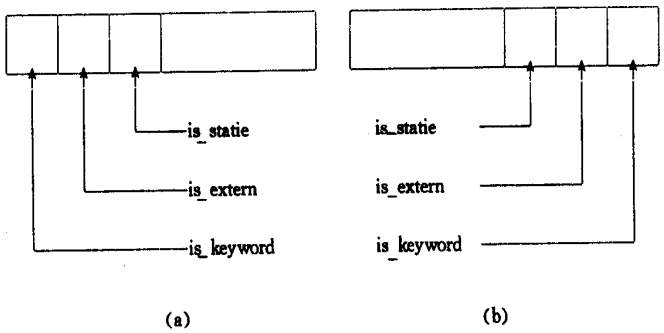


图10-2 值的分配

```
insword.oper1 = 6 ;
insword.oper2 = 3 ;
if (flags.is_keyword == 1)
    i ++ ;
```

有了位段我们就可以按名去访问所需要的位,而不必了解这些位在一个字中的具体位置。

在位段结构中可以定义无名位段,这种位段不能在程序中被访问它们只起占位分隔的作用。如:

```
struct iden_flags {
    unsigned is_keyword: 1 ;
    unsigned : 7 ;
    unsigned is_extern : 1 ;
    unsigned : 6 ;
    unsigned is_static : 1 ;
} flgs ;
```

这时 flgs 中位段的情况如图 10.3 所示。(假定从左到右分配位段)

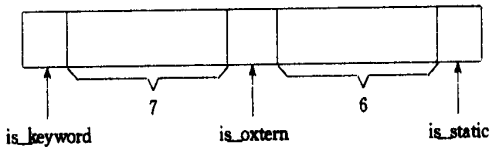


图10-3 无名位段

这种无名位段在描述寄存器时是很有用的,因为一些状态寄存器只用到部分位,而且这些位也不一定连续。

关于位段,最后再说明几点。

1. 每个位段不能超过一个字长(int 的长度),这一点应该是很明白的。

2. 位段不能跨越两个字。也就是说位段只能在一个字中,如果一个字中剩下的空间不能存放一个位段,那么这个位段就应该放到下一个字中。

3. 实际中位段总是无符号量,只能存于机器中,位段没有地址,不能对位段使用指针,也不能定义位段数组。

4. 引用位段相当于引用一个无符号整型量。

第十一章　输入输出与文件操作

本章学习重点

本章介绍C语言的输入输出函数，以及两种形式的文件系统：缓冲文件系统和非缓冲文件系统。通过本章的学习应能熟练而灵活地使用C语言的输入输出函数，掌握缓冲文件的使用，并对非缓冲文件有所了解。

第一节　输入输出函数

C语言本身并不提供输入输出语句，而是用一组库函数来实现数据的输入输出。不同的系统通常提供不同的输入输出函数，但有些常用的是各系统中共同的。现就介绍一下这些最基本的输入输出库函数。

11.1.1　字符输入输出

C语言标准库提供了一次输入一个字符的函数getchar和一次输出一个字符的函数putchar。

getchar的引用形式为

ch = getchar()

用于从标准输入设备(终端)上读入并返回一个字符。

putchar函数的引用形式为

putchar(ch)

它把字符ch送到标准输出设备(终端屏幕)上去。一般我们可以把ch定义为一个字符型量，但有时定义成整型量更好，如例1.4中的

```
int  c ;
while ( ( c = getchar() ) ! = EOF)
    putchar(c) ;
```

由于EOF定义为−1，上面的写法比写成

```
char  c ;
while ( ( c = getchar() ) ! = EOF)
    putchar(c) ;
```

具有更好的通用性，因为在一些系统中char型永远取正值。

putchar可以用来输出字符变量的值，也可以用来直接输出一个字符常数，如

```
putchar('a')
putchar('\n')
```

还可以输出用三位八进制表示的字符值，如

```
putchar('\101')
```

将字母'A'输出。

在使用 getchar 和 putchar 之前不要忘记在程序的前面写上

#include "stdio.h"

实际上 getchar 和 putchar 是作为宏在 stdio.h 中定义的。

11.1.2 字符串输出输出

有时候我们希望一次输入或输出一个完整的字符串,C 库函数提供了 gets 和 puts 用来完成这个工作。gets 接收来自标准输入的一个字符串,puts 发送一个字符串到标准输出。

gets 通常的调用形式为:

gets (str)

其中 str 是字符指针,下面的程序读入一个字符串到字符数组,并输出它的长度。

例 11.1

```
main()
{
    char  str[80];
    gets(str);
    printf("%d\n", strlen(str));
}
```

输入

```
how  are  you
```

得到的输出值

```
11
```

从这个例子中我们可以看到,gets 接收字符串直到遇到回车符'\n'时为止,并以一个'\0'代替'\n',这与 scanf("%s", str);得到的结果是不同的,scanf 遇到空白符(空格,制表和换行符)都停止接收后面的字符。

函数 puts 通常的引用形式为

puts(str)

它将字符指针 str 所指的字符串输出到标准输出设备上。它就相当于

```
printf("%s\n", str);
```

11.1.3 格式输出

格式化输入输出用函数 scanf 和 printf 来完成。实际上我们从本书的开始就一直在使用它们,只是没有做细致的讨论。下面我们先来看一下格式输出 printf。

格式输出函数 printf 的一般形式为

printf(控制字符串,参数 1,参数 2…)

它的功能是,在控制字符串的控制下,将参数转换为规定的格式在标准输出设备(屏幕)上输出。控制字符串可包含两类内容,第一类是普通字符,这些字符只被简单地复制到屏幕上,第二类是格式命令。格式命令以百分号%开头,后跟格式转换字符,每个格式命令导致 printf 中对应参数的转换和输出。表 11.1 给出了 printf 中可用的转换字符及其含义。

表 11.1 printf 中使用的转换字符

转换字符	含　义
d	以十进制形式输出整数值
o	以八进制形式输出整数值
x	以十六进制形式输出整数值
u	以无符号数形式输出整数值
c	输出字符值
s	输出字符串
f	输出十进制浮点数
e	以科学表示法输出浮点数
g	等效%f 或%e,输出两者占位较短的

除转换字符外,C 语言还特地为输出 long 型数据提供了一个长度修正符 l,它与转换字符 d(或 u)连用,形式为%ld(或%lu)表示输出值的类型是 long(或 unsigued long),而不是 int(或 unsigned)。由于 C 语言没有直接提供输出 double 型数据的转换字符,有些机器系统也可用 l 与 f 或 e 连用,用来输出 doulble 型数据,以期获得较长的精确位。

在格式命令的%和转换字符中间可以加上一些任选的说明符用来对输出格式做进一步的限定。这些说明符在表 11.2 中列出:

表 11.2 printf 中的任选说明符

符　号	含　义
-	输出时左对齐,缺省时表示右对齐,如 printf("%-10d", i);
dd	指定对应参数输出时所占的最小域宽,数据长度不足时,用空格补齐,如 printf("%20s", str);

dd.dd	用于浮点数或字符串输出,用于浮点数时,小数点前面的数表示浮点数所占的位数(包括小数点),小数点后面的数表示精确到小数点后第几位,如
	printf("%10.4f", x);
	用于字符串时,小数点前面的数表示输出所占的位数,小数点后的数表示实际要输出的字符个数,如
	printf("%7.5s", str);

了解输出格式的最好办法是通过例子来看一下各种数据的输出结果。

例 11.2

```
main()
{
    char  c = 'a';
    static  char  str[] = "see  you";
    int  i = 1234;
    float  x = 123.456789;
    float  y = 1.2;
    printf ("1:|%c|%s|%d|%f|%e|%f|\n", c, str, i, x, x, y);
```

```
    printf ("2:|%4c|%10s|%6d|%12f|%15e|%10f|\n", c, str, i, x, x, y) ;
    printf ("3:|%-4c|%-10s|%-6d|%-12f|%-15e|%-10f|\n", c, str, i, x,
        x, y) ;
    printf ("4:|%0c|%6s|%3d|%9f|%10e|%2f|\n", c, str, i, x, x, y) ;
    printf ("5:|%12.2f||%12.0f|%\n", x, x) ;
    printf ("6:|%.2f|\n", x) ;
            printf("7:|%10.4f|\n", y) ;
    printf ("8:|%8.3s|%8.1s|%8.0s|\n", str, str, str) ;
}
```

运行结果

```
1:|a|see you|1234|123.456787|1.23457e+0.2|1.200000|
2:|   a| see you|  1234|  123.456787|   1.23457e+0.2| 1.200000|
3:|a |see you   |1234  |123.456787  |1.23457e+0.2   |1.200000 |
4:|a|see you|1234|123.456787|1.23457e+0.2|1.200000|
5:|       123.46|          123|
6:|123.46|
7:|    1.20000|
8:|     see|       s|          |
```

第一行输出没有指定域宽，则输出结果所占空间按数值所占空间安排(浮点数小数点后占 6 位)，细心的者可能发现用%f 输出浮点数时，数据是不精确的，如 123.456789 输出为 123.456787。这是因为在机器内部表示浮点数时没有足够的位数来保证精确存储 9 位有效数据，所以输出的结果是不精确的。同样用%e 也只显示了部分精度的数据(最后 1 位被 4 舍 5 入)。

第二行输出由于指定的域宽超过实际的需要，所以增加了空格以补足所要占的位数，没有符号“-”，输出全部为右对齐。

第三行输出由于有符号“-”，所以全部左对齐。

第四行输出由于指定的域宽小于数值输出实际所占的位数，系统忽略指定的域宽，所以输出与第一行一样。

第五行输出由于指定的小数点位数小于实际的位数，系统按要求截取小数点后的位数并做 4 舍 5 入。

第六行输出由于只指定了小数点后的精度，所以系统只截取小数点的位数，并不为你安排空格。

第七行输出按照要求截取小数点位数，由于长度不够要求的域宽，所以在前面补足空格。

第八行输出按你的要求(小数点后面的数)输出字符个数。

11.1.4 格式输入

格式输入用 scanf 函数实现，其一般形式为

(控制字符串，参数 1，参数 2…)

它能读入各种类型的数据，并自动将其转换为恰当的形式存放于由参数指示的变量中。这时

参数必须是指针，以指向变量的存储位置。与 printf 类似，scanf 的控制字符串中也可以有多种转换字符，如表 11.3 所列出的那样。

表 11.3　scanf 中的转换字符

转换字符	含　义
d	期待输入十进制整数
o	期待输入八进制整数
x	期待输入十六进制整数
h	期待输入短整型数
c	期待输入一个字符
s	期待输入一个字符串
f	期待输入一个浮点数
e	期待输入一个浮点数

与 printf 一样，在想输入 long 型值时，可用 l 与 d 相连结，如%ld(也可用%lo、%lx，但极少有人用)，有些系统允许用%lf 或%le 输入 double 型数据。

在格式控制字符串中还可以包括：

(1)空白符：它们无意义，被忽略。

(2)普通字符：在输入数据时必须输入与这些字符相同的字符，如

```
scanf("a=%d,b=%d", &a, &b) ;
```

输入时应键入

```
a=10,b=5
```

如不打入 a=，和 b=系统则等待你输入，这种形式的输入你绝不能交给用户使用，因为别人怎么知道你们源程序中是怎样写的呢？

(3)域宽：放在%与转换字符之间的整数，用来指明输入数据的位数。

scanf 中控制字符串后的各参数都应是指针型，因此除了转换字符用 s，参数用数组或指向字符的指针外，其它转换字符对应的各种非指针变量前都应该加上取地址运算符 &，除非你又用一个指针指向存放数据的变量，并间接地用指针接收数据(实际中很少有人这么做)。

当 scanf 从输入字符流中获取各种类型的数据时，它是怎么知道什么时候数据结束的呢？给出域宽是一个办法，但这很少使用，更一般的方法可以通过下面的例子来了解一二。

例 11.3

```
main()
{
    char  ch ;
    int  i ;
    char  str[80] ;
    float  x ;
    scanf("%c%d%s%f", &ch, &i, str, &x) ;
    printf("%c  %d  %s  %f\n", ch, i, str, x) ;
}
```

当输入为

w 123 hello 123.456

时输出为

w 123 hello 123.456001

现在把输入值的方式改一下

w

123

hello

123.456

输出仍然是

w 123 hello 123.456001

也就是说我们可以用空格符或回车符(还有制表符 TAB 键)将输入的字符隔开,这样 scanf 就能从输入流中获取所需的数据。

我们再把输入的方式改一下

w123hello123.456

回车后没有输出。再打入 1,这时的输出结果为

w 123 hello123.456 1.000000

这是为什么呢?原来,scanf 根据转换字符的含义从输入字符流中获取数据,当输入流中的数据类型与转换字符所期待的字符类型不符时,就认为当前数据结束,于是%c 只读入一个字符将 w 读入 ch,%d 格式接收数字,当遇到字母 h 时认为整数值结束,将 123 送给 i,%s 接收各种字符直到遇到空白符(空格、制表、回车)时才认为当前数据结束,所以将 hello 与 123.456 一起读进了 str。

应当注意的是%c 可以接收各种字符,也包括空格、回车一类字符,所以当输入

↓ (回车)

123 hello 123.456

时,输出为

123 hello 123.456001

也就是先输出了一个回车符(相当于空一行)然后再输出其他各值。

第二节 缓冲文件系统

在前面的程序中,我们所需要输入的数据都来自标准输入(终端键盘),所得到的结果也总是送到标准输入设备(屏幕)上去。而实际许多场合中,我们需要从文件读取数据,或者把数据送到某个文件中存起来,这就要用到文件的输入输出。

C 语言把文件看成是一个字符(字节)序列。即文件是由一个一个字符(字节)的数据顺序组成。根据数据的组织形式,可以把文件分为两类:文本文件和二进制文件。文本文件的每个字节存放一个 ASC Ⅱ 码,代表一个字符,二进制文件中的数据以数据在内存中的形式存放。

在传统的 UNIX 标准中，用缓冲文件系统处理文本文件，而用非缓冲文件系统处理二进制文件。所谓缓冲文件系统是指：系统自动地为每个正在使用的文件开辟一个缓冲区，从内存向磁盘存数据或从磁盘向内存取数据都通过这个缓冲区。而非缓冲文件系统是指：系统不自动为文件开辟缓冲区，而由程序自己为所需的文件开辟缓冲区。由于这两种文件系统中有许多功能是重叠的，因此 ANSI C 建议只保留缓冲文件系统，并扩展了它的功能。这样在 ANSI C 中，缓冲文件既用于处理文本文件，又用来处理二进制文件。下面我们就来讨论缓冲文件系统，而把非缓冲文件系统留在下节。

11.2.1 打开和关闭文件

打开文件用函数 fopen 完成，其一般形式为

```
FILE  *fp ;
fp = fopen(文件名, 文件使用方式);
```

这里的 FILE 是一个存放文件属性的结构，它在 stdio.h 中定义，我们把 fp 定义为指向文件结构的指针，就可以通过 fp 来操作文件了。函数 fopen 以指定的方式打开文件，如

```
fp = fopen("file1", "r") ;
```

以读方式打开文件 file1。文件使用方式在表 11.4 中列出。

表 11.4 文件使用方式

方式	含义
"r"	为输入打开一个文本文件
"w"	为输出打开一个文本文件
"a"	向文本文件尾部添加
"rb"	为输入打开一个二进制文件
"wb"	为输出打开一个二进制文件
"ab"	向二进制文件尾部添加
"r+"	为读写打开一个文本文件
"w+"	为读写产生一个文本文件
"a+"	为读写打开一个文本文件
"rb+"	为读写打开一个二进制文件
"wb+"	为读写产生一个二进制文件
"ab+"	为读写打开一个进进制文件

以"r"打开的文件只能用于读，而以"w"打开的文件只能用于写，如果这个文件不存在就创建这个文件，如果文件已存在则"w"方式打开文件将使原来文件的内容全部丢失。如果要想在文件的末尾加新的数据，就要以"a"方式打开文件。你不能直接在文件的中间插入数据。如果一个文件由于某种原因打不开，如读一个不存在的文件，fopen 将返回一个 NULL，表示打开文件失败，所以打开文件最常见的操作形式为

```
if ( (fp = fopen("file1", "r") ) == NULL) {
    printf("can  not  open  file.\n") ;
    exit(1) ;
}
```

这里 exit 函数的功能是关闭所有打开的文件并强迫程序结束。一般 exit 带参数值 0 表示正常结束，带非 0 值表示出错后结束，操作系统中可以接收返回的参数值。

文件的关闭很简单，只需要写

fclose(fp)；

这里 fp 是 fopen 返回的指向文件结构的指针。

11.2.2 文件的读写

文件打开之后就可以进行读写操作了。C 语言提供了多种用于读写文件的函数，这里我们只介绍几种最常用的。

1. getc 和 putc

getc 的引用形式为

ch = getc(fp)；

这里 fp 是调用 fopen 时的返回值。getc 从 fp 所指的文件中读取一个字符返回，当读到文件尾时返回文件结束标志 EOF。

putc 的调用形式为

putc(ch, fp)；

这里的 fp 是调用 fopen 时的返回值，putc 将字符 ch 写到 fp 向指的文件中去。

例 11.4　文件拷贝

在操作系统中，我们经常用到文件拷贝命令，其最简单的形式为

copy　from　to

现在我们就编一个程序来完成这个功能。很明显，这里需要用到命令行参数，程序如下：

```
#include "stdio.h"
main(arge, argv)
int  argc ;
char  *argv[] ;
{
    FILE  *fin , *fout ;
    int  ch ;
    if (argc ! = 3) {
        printf("Usage:copy  from  to\n") ;
        exit(1) ;
    }
    if ( (fin = fopen(argv[1], "r") ) == NULL) {
        printf("can  not open  %s\n", argv[1]) ;
        exit(1) ;
    }
    if ( (fout = fopen(argv[2], "w") ) == NULL) {
        printf("can  not  open  %s\n", argv[2]) ;
        exit(1) ;
    }
```

```
    while ( (ch = getc(fin) ) ! = EOF)
        putc(ch,fout) ;
    fclose(fin) ;
    fclose(fout) ;
}
```

这个程序可以用于文本文件的复制，如果复制二进制文件应该怎样改呢?当然打开文件的方式要改成“rb”和“wb”，但仅此并不能保证程序的正确执行。我们知道，ASCII 字符不可能取负值，因此一但读到 EOF(值为－1)就可以断定文件已经读完，但在二进制文件中，一个字节的值可能正是全 1，它就相当于十进制数－1，这时如果还用 EOF 做判定条件就可能只复制了部分文件内容，为了解决这个问题，ANSIC 增加了一个专门用于判断文件结束的函数 feof，当文件结束时它返回 1，否则返回 0，这样，复制二进制文件就可以用下面的语句实现。

```
while (! feof(fp) ) {
    ch = getc(fp) ;
    putc(ch, fp) ;
}
```

以代替例 11.4 中的

```
while ( (ch = getc(fp) ) ! = EOF)
    putc(ch, fp) ;
```

2. fscanf 和 fprintf

fscanf 和 fprintf 很像 scanf 和 printf，只是它们对文件进行读写，而不是对标准输入输出设备进行读写。

fscanf 的一般形式为：

fscanf(fp,控制字符串,参数 1,参数 2…) ;

fprintf 的一般形式为

fprintf(fp,控制字符串,参数 1,参数 2…) ;

3. fread 和 fwrite

这两个函数是 ANSIC 新增加的，它们可以读写一个数据块，它们的一般调用形式为

fread(buffer,size,count,fp) ;

fwrite(buffer,size,count,fp) ;

其中：

buffer 是一个指针，对 fread 而言，它指向存放读入数据的缓冲区，对于 fwrite 而言，它指向要写入文件的数据所存放的缓冲区。

size 是一次要读写的字节数。

count 是要读写多少项。

fp 是指向文件的指针。

如果文件以二进制形式打开，则用 fread 和 rwrite 可以读写任何类型的数据，如

```
fread(&f, sizeof(float), 1, fp) ;
```

从 fp 指向的文件中读一个 float 型值到变量 f 中。从这种形式上看，我们也可以用 fread 和

fwrite 读写一个结构。实际上，这正是 fread 和 fwrite 最主要的用途之一。如有结构

```
struct  struct_type {
    ⋮
} struct_var ;
```

则读一个结构值可以写

```
fread(&struct_var, sizeof(struct_type), 1, fin) ;
```

类似地，写一个结构值可以用

```
fwrite(&struct_var, sizeof(struct_type), 1, fout) ;
```

11.2.3 标准输入输出文件

C程序开始执行时，系统自动打开三个文件，它们是：标准输入(stdin)，标准输出(stdout)和标准错误输出(stderr)。通常这些文件是对终端来说的，前面介绍的 getchar 和 putcha 不过是 getc 和 putc 的特例，在 stdio.h 中，它们被定义为宏：

```
#define  getchar  getc(stdin)
#define  putchar(c)  putc(c, stdout)
```

同样，printf 和 scanf 也可以用

```
fprintf(stdout, 控制字符串, 参数 1, 参数 2…)
```

和

```
fscanf(stdin, 控制字符串, 参数 1, 参数 2…)
```

来代替。

通常，stdin 用来从终端键盘上读，stdout 用做向终端屏幕上写，stderr 也是在屏幕上显示。在支持 I/O 重定向的操作系统中，stdin 和 stdout 可以被重定向，而 stderr 仍然是在终端屏幕上显示输出。如在 UNIX 系统或 DOS 系统下，我们可以在命令行中写下命令

```
file < indata
```

这时 file 中所有的 getchar 和 scanf 等就不再是从键盘上接收数据，而是从文件 indata 中读取数据，这就是输入重定向。而写下命令

```
file < outdata
```

则 file 中所有的 putchar 和 printf 等都把输出结果写到文件 outdata 中，这就是输出重定向。也可以输入输出同时重定向，如

```
file < indata > outdata
```

由于 stdout 可以被重定向，所以一个程序中的错误信息最好用下面形式输出

```
fprintf(stderr, 控制字符串, 参数 1, 参数 2…) ;
```

以确保它显示在屏幕上，如

```
fprintf(stderr, "can  not  open  file\n") ;
```

11.2.4 文件的随机访问

我们并不总是希望按顺序读写文件，有时也需要在文件的任意指定位置读写文件，这就是文件随机读写的概念。C 语言中用 fseek 实现这个功能，fseek 的一般调用形式为

fseek(fp, offset, origin) ;

其中

fp 是指向文件的指针。

origin 是起始点，它可以取三个值，为 0 时表示从文件起始位置开始，为 1 时表示从文件当前位置开始，为 2 时表示从文件尾部开始。

offset 是一个位移量，它表示从起始点(origin)移动的字节数。它是一个 long 型数据，这样保证当文件长度超过 64k 时不致于出问题。

fseek 调用成功时返回 0，调用失败时返回非 0 值。

下面是几个 fseek 函数调用的例子。

```
fseek(fp, 20L, 0) ; /* 将位置指针定位于距文件头 20 字节处 */
fseek(fp, -2L, 1) ; /* 将位置指针定位于当前位置的前 2 字节处 */
fseek(fp, -10L, 2) ; /* 将位置指针定位于文件尾前面 10 字节处 */
```

有了 fseek 就可以进行随机读写了。下面的例子将文件 data 中的内容反序输出。

例 11.5

```
#include  "stdio.h"
main()
{
    FILE  *fp ;
    if (fp = fopen("data", "r") ) ! = NULL) {
        fprintf(stderr, "can  not  open  file.\n") ;
        exit(1) ;
    }
    fseek(fp, -3L, 2) ;/* 将位置指针置于文件中最后一个有效字符处，*/
                       /* 适用于 TurboC */
    putchar(getc(fp) ) ;
    while (fseek(fp, -2L, 1) == 0) /* 倒退一个字符 */
        putchar(getc(fp) ) ;
}
```

程序中位移量用 -2L 是因为每次调用 getc 使位置指针向前移动了一个字节，用 -1L 只能回到原来的位置，而用 -2L 才真正是后退了一个字节。

第三节　非缓冲文件系统

非缓冲文件系统是基于 UNIX 的，ANSI 标准不包含这部分内容。这意味着以后有些 C 可能不允许使用与之有关的一些函数。但由于 C 与 UNIX 的渊源，早期的 C 程序大量使用了非缓冲文件系统中的函数，为了能够继续使用这些程序，至少也为了能读懂这些程序，我们还是应该了解非缓冲文件系统。

下面我们就非缓冲文件系统中的一些常用函数作个介绍。

1. open 函数

open 函数用于打开文件，其调用形式为：

int fd ;

fd = open(文件名, 打开模式) ;

open 用返回一个较小的整数 fd(称为文件描述字)来标识打开的文件,当打开失败时返回－1。打开模式可以取三种值:0 表示为读打开,1 表示为写打开,2 表示为读写打开。

2. creat 函数

creat 函数用来创建一个新文件,如果文件已经存在,则 creat 使文件中原来的内容全部丢失,其调用方式因系统而异,TurboC 的调用形式为:

fd = creat(文件名, 打开格式) ;

当文件创建失败时返回－1。

3. close 函数

close 用于关闭文件,其一般形式为:

close(fd) ;

fd 是调用 creat 或 open 时返回时文件描述字。

4. read 函数

read 的作用是从指定的文件中读取若干个字符到程序开辟的缓冲区,其一般调用形式为:

n = read(fd, buf, count) ;

这里 fd 是文件描述字,buf 是指向用户开辟的缓冲区的指针,count 是要求读入的字节数,read 返回真正读入的字节数,文件结束时返回 0,出错时返回－1。

5. write 函数

write 用于从指定的内存缓冲区将若干个字节的内容写到指定的文件中去,其一般调用式为:

n = write(fd, buf, count) ;

其中 fd 是文件描述字,buf 是指向存放要输出数据的缓冲区的指针,count 是要输出的字节数。返回值应与 count 相同,不同时意味着有某种错误出现。

在使用非缓冲文件系统时,也有三个文件被自动打开,它们是标准输入、标准输出和标准错误输出,其 fd 分别为 0、1、2。

下面我们通过一个例子来看一下非缓冲文件的使用。

例 11.6 显示文件的内容。

```
#define  SIZE  512
main(argc, argv)
int  argc ;
char  *argv[] ;
{
    char  buf[SIZE] ;
    int  fd,n ;
    if (argc ! = 2) {
        printf("Usage:type  file.\n") ;
        exit(1) ;
```

```
    }
    if ( (fd = open(argv[1], 0) ) == -1) {
        printf("can not open %s\n", argv[1]) ;
        exit(1) ;
    }
    while ( (n = read(fd, buf, sizeof(buf) ) > 0)
        write(1, buf, n) ;
    exit(0) ;
}
```

这个程序中 read 函数返回值可能与 sizeof(buf)不等，所以输出时只能用 n，而不能用 sizeof(buf)表示输出的字节数。函数调用 exit(0)关闭所有打开的文件，然后结束程序运行。

附录一、ASCII 码表

ASCII 值	字符	ASCII 值	字符
000	(null)	034	"
001		035	#
002		036	$
003		037	%
004		038	&
005		039	'
006		040	(
007	(bell)	041	)
008		042	*
009	(tab)	043	+
010	(line feed)	044	,
011	(home)	045	-
012	(form feed)	046	.
013	(carriage return)	047	/
014		048	0
015		049	1
016		050	2
017		051	3
018		052	4
019		053	5
020		054	6
021		055	7
022		056	8
023		057	9
024		058	:
025		059	;
026		060	<
027		061	=
028		062	>
029		063	?
030		064	@
031		065	A
032	(space)	066	B
033	!	067	C

ASCII 值	字符	ASCII 值	字符
068	D	098	b
069	E	099	c
070	F	100	d
071	G	101	e
072	H	102	f
073	I	103	g
074	J	104	h
075	K	105	i
076	L	106	j
077	M	107	k
078	N	108	l
079	O	109	m
080	P	110	n
081	Q	111	o
082	R	112	p
083	S	113	q
084	T	114	r
085	U	115	s
086	V	116	t
087	W	117	u
088	X	118	v
089	Y	119	w
090	Z	120	x
091	[	121	y
092	\	122	z
093	]	123	{
094	^	124	|
095	—	125	}
096	`	126	~
097	a	127	DEL

附录二、C 语言中的关键字

auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	int	long	register	return
short	signed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while			

说明：

关键字是系统用于专门用途的标识符，用户不能将它们当作自己的变量名、数组名、文件名、类型名等。

附录三、运算符和结合性

优生级	运算符	含义	要求运算对象个数	结合方向
1	() [] → .	圆括号 下标运算符 指向结构成员运算符 结构成员运算符		自左至右
2	! ~ ++ -- - (类型) * & sizeof	逻辑非运算符 按位取反运算符 自增运算符 自减运算符 负号运算符 类型转换运算符 指针运算符 地址运算符 长度运算符	1 (单目运算符)	自右至左
3	* / %	乘法运算符 除法运算符 求余运算符	2 (双目运算符)	自左至右
4	+ -	加法运算符 减法运算符	2	自左至右
5	<< >>	左移运算符 右移运算符	2	自左至右
6	<<=>>=	关系运算符	2	自左至右
7	== !=	等于运算符 不等运算符	2	自左至右
8	&	按位与运算符	2	自左至右
9	^	按位异或运算符	2	自左至右
10	|	按位或运算符	2	自左至右
11	&&	逻辑与运算符	2	自左至右
12	||	逻辑或运算符	2	自左至右
13	?:	条件运算符	3 (三目运算符)	自右至左

优生级	运算符	含义	要求运算对象个数	结合方向
14	= += -= *= /= %= >>= <<= &= ^= |=	赋值运算符	2	自右至左
15	,	逗号运算符 (顺序求值运算符)		自左至右

说明：

(1)同一优先级的运算符优先级别相同，运算顺序由结合方向决定。例 如，*与/具有相同的优先级别，其结合方向自左至右，因此 3 * 5/4 相当于(3 * 5)/4。－与＋＋为同一优先级，结合方向从右至左，因此－i＋＋相当于－(i＋＋)。

(2)不同的运算符要求的运算对象数目不同，如＋(加)和－(减)为双目运算符，要求运算符两侧各有一个运算对象(如 6＋2、7－3、8/5 等)。而＋＋和－(负号)等一元运算符，只能在运算符的一侧出现一个运算对象(如－a、i＋＋、j－－、(float)k、size of(int)、* p 等)。条件运算符?:是 C 中唯一的一个三目运算符，如 x? a:b。